Accurate Power Monitoring at Operating System Level towards Energy-Aware Android Devices

BY
EMANUELE DEL SOZZO
B.S., Politecnico di Milano, Milan, Italy, July 2012

THESIS

Submitted as partial fulfillment of the requirements for the degree of Master of Science in Computer Science in the Graduate College of the University of Illinois at Chicago, 2015

Chicago, Illinois

Defense Committee:
Jon A. Solworth, Chair and Advisor
Jakob Eriksson
Marco D. Santambrogio, Politecnico di Milano
ACKNOWLEDGEMENTS

Many people made this thesis possible: they gave me help, support, suggestions, guidance, inspiration, patience, precious time, confidence and everything I needed to produce this work. First of all, I want to thank Matteo Ferroni and Alessandro Nacci, who continued to motivate and believe in me in every moment.

A big thank you to my advisor, Marco Domenico Santambrogio: he supported and trusted in me from the beginning to the end of my work, particularly when I needed his help the most.

I want to thank my family, who gave me such a big chance to study at UIC. My family patience and support helped me concentrating on my work.

A special thanks to Andrea Brancaleoni, Stefano Cagninelli, Andrea Damiani, Alberto Scolari and Francesco Trovò. Without their fundamental help and knowledge, this work would not have been possible.

A thank you also to Prof. Jon Solworth, my UIC advisor, who gave me the opportunity to work as his research assistant.

Thank you also to all the people in the NECSTLab at Politecnico di Milano, with whom I shared the working time in a funny and stimulating atmosphere: helping us each other and sharing our knowledge are really a pleasure.

A thank you to the PoliMI-UIC students of Fall 2013, who shared with me such a great experience. In particular Enrico Deiana and Luca Graziani, who shared with me also the Spring 2014 semester.
ACKNOWLEDGEMENTS (Continued)

Finally, I want to thank Giuseppe De Nittis, Leone D’Emilio and Stefano Campanella, who shared with me innumerable hours of study and funny moments during these years of “Poli”.

EDS
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Context definition</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Power monitoring for mobile devices</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Daemon approach motivations</td>
<td>3</td>
</tr>
<tr>
<td>2 BACKGROUND</td>
<td>5</td>
</tr>
<tr>
<td>2.1 Android</td>
<td>5</td>
</tr>
<tr>
<td>2.2 Android features and characteristics</td>
<td>7</td>
</tr>
<tr>
<td>2.3 Android application components</td>
<td>11</td>
</tr>
<tr>
<td>2.4 Android worldwide diffusion</td>
<td>13</td>
</tr>
<tr>
<td>2.5 Android choice</td>
<td>15</td>
</tr>
<tr>
<td>3 STATE OF THE ART</td>
<td>18</td>
</tr>
<tr>
<td>3.1 Power management in computers</td>
<td>19</td>
</tr>
<tr>
<td>3.2 Power models for mobile devices</td>
<td>21</td>
</tr>
<tr>
<td>3.2.1 Battery draining factors</td>
<td>22</td>
</tr>
<tr>
<td>3.2.2 Approaches to the problem</td>
<td>23</td>
</tr>
<tr>
<td>3.3 Power consumption measurement methodologies</td>
<td>24</td>
</tr>
<tr>
<td>3.3.1 Application level</td>
<td>25</td>
</tr>
<tr>
<td>3.3.2 User level</td>
<td>27</td>
</tr>
<tr>
<td>3.3.3 System level</td>
<td>29</td>
</tr>
<tr>
<td>3.3.3.1 External measurements</td>
<td>30</td>
</tr>
<tr>
<td>3.3.3.2 Only APIs</td>
<td>33</td>
</tr>
<tr>
<td>3.3.3.3 Custom and Internal Measurements</td>
<td>35</td>
</tr>
<tr>
<td>3.3.4 Results of the survey</td>
<td>36</td>
</tr>
<tr>
<td>3.4 Power consumption Apps</td>
<td>38</td>
</tr>
<tr>
<td>3.4.1 Snapdragon™ BatteryGuru</td>
<td>38</td>
</tr>
<tr>
<td>3.4.2 Battery Doctor</td>
<td>38</td>
</tr>
<tr>
<td>3.4.3 Android L</td>
<td>41</td>
</tr>
<tr>
<td>4 PROBLEM DEFINITION AND PROPOSED SOLUTION</td>
<td>45</td>
</tr>
<tr>
<td>4.1 Problem definition</td>
<td>45</td>
</tr>
<tr>
<td>4.2 MPower</td>
<td>48</td>
</tr>
<tr>
<td>4.2.1 MPower features</td>
<td>48</td>
</tr>
<tr>
<td>4.2.1.1 MPower components</td>
<td>51</td>
</tr>
<tr>
<td>4.2.2 Power Model Visualization</td>
<td>54</td>
</tr>
<tr>
<td>4.3 Proposed Solution</td>
<td>55</td>
</tr>
<tr>
<td>CHAPTER</td>
<td>PAGE</td>
</tr>
<tr>
<td>---------</td>
<td>------</td>
</tr>
<tr>
<td>5</td>
<td>IMPLEMENTATION</td>
</tr>
<tr>
<td>5.1</td>
<td>Context background</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Power management in Android</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Android Architecture</td>
</tr>
<tr>
<td>5.1.2.1</td>
<td>Android Kernel</td>
</tr>
<tr>
<td>5.1.2.2</td>
<td>Binder</td>
</tr>
<tr>
<td>5.1.2.3</td>
<td>Dalvik</td>
</tr>
<tr>
<td>5.1.2.4</td>
<td>System Services</td>
</tr>
<tr>
<td>5.2</td>
<td>Rooting</td>
</tr>
<tr>
<td>5.2.1</td>
<td>How to root</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Custom ROM: the CyanogenMod</td>
</tr>
<tr>
<td>5.3</td>
<td>Daemon logger implementation</td>
</tr>
<tr>
<td>5.4</td>
<td>Integration with MPower</td>
</tr>
<tr>
<td>6</td>
<td>RESULTS</td>
</tr>
<tr>
<td>6.1</td>
<td>MPower OS level tests</td>
</tr>
<tr>
<td>6.1.1</td>
<td>Target devices</td>
</tr>
<tr>
<td>6.1.2</td>
<td>Case studies</td>
</tr>
<tr>
<td>6.1.3</td>
<td>Results of MPower tests</td>
</tr>
<tr>
<td>6.2</td>
<td>Monsoon Power Monitor</td>
</tr>
<tr>
<td>6.3</td>
<td>Battery impact comparison</td>
</tr>
<tr>
<td>6.4</td>
<td>Daemon logging frequency</td>
</tr>
<tr>
<td>7</td>
<td>CONCLUSIONS AND FUTURE WORKS</td>
</tr>
<tr>
<td>7.1</td>
<td>Contributions and limits</td>
</tr>
<tr>
<td>7.2</td>
<td>Current works</td>
</tr>
<tr>
<td>7.3</td>
<td>Future works</td>
</tr>
<tr>
<td>CITED LITERATURE</td>
<td>104</td>
</tr>
<tr>
<td>VITA</td>
<td>110</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>TABLE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>7</td>
</tr>
<tr>
<td>II</td>
<td>14</td>
</tr>
<tr>
<td>III</td>
<td>37</td>
</tr>
<tr>
<td>IV</td>
<td>53</td>
</tr>
<tr>
<td>V</td>
<td>74</td>
</tr>
<tr>
<td>VI</td>
<td>76</td>
</tr>
<tr>
<td>VII</td>
<td>79</td>
</tr>
<tr>
<td>VIII</td>
<td>85</td>
</tr>
<tr>
<td>IX</td>
<td>87</td>
</tr>
<tr>
<td>X</td>
<td>91</td>
</tr>
<tr>
<td>XI</td>
<td>98</td>
</tr>
</tbody>
</table>

- I ANDROID VERSIONS DIFFUSION
- II WORLDWIDE MOBILE OPERATING SYSTEMS DIFFUSION
- III COMPONENTS CONSIDERED BY PAPERS IN THIS SURVEY
- IV INSTANCE OF TTL TABLE
- V AOSP CONTENT SUMMARY
- VI SOME OF MODULE BUILD TEMPLATES
- VII DATA COLLECTED BY THE DAEMON
- VIII COMPARISON BETWEEN THE TESTING DEVICES
- IX CASE STUDIES COMPARISON
- X HYPOTHESIS TEST RESULTS
- XI SOME SIGNIFICANT TESTS
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Data collected during a 7-day period ending on November 3, 2014</td>
</tr>
<tr>
<td>2</td>
<td>Android software stack[1]</td>
</tr>
<tr>
<td>4</td>
<td>Number of Apps available in the top App Stores[3]</td>
</tr>
<tr>
<td>5</td>
<td>Battery statistics on LG Nexus 5 running Android K</td>
</tr>
<tr>
<td>6</td>
<td>BatteryGuru screenshots</td>
</tr>
<tr>
<td>7</td>
<td>Battery Doctor screenshots</td>
</tr>
<tr>
<td>8</td>
<td>Battery historian</td>
</tr>
<tr>
<td>9</td>
<td>Inter-process method invocation</td>
</tr>
<tr>
<td>10</td>
<td>MPower screens</td>
</tr>
<tr>
<td>11</td>
<td>MPower internal structure</td>
</tr>
<tr>
<td>12</td>
<td>Android Power Management Application Framework</td>
</tr>
<tr>
<td>13</td>
<td>Android system states</td>
</tr>
<tr>
<td>14</td>
<td>Android architecture</td>
</tr>
<tr>
<td>15</td>
<td>Android System Services</td>
</tr>
<tr>
<td>16</td>
<td>Summary of App-service interaction</td>
</tr>
<tr>
<td>17</td>
<td>Example of ClockworkMod recovery screen</td>
</tr>
<tr>
<td>18</td>
<td>Android build system[4]</td>
</tr>
<tr>
<td>19</td>
<td>Average discharging curves in the multimedia case study</td>
</tr>
<tr>
<td>20</td>
<td>Average discharging curves in the calling case study</td>
</tr>
<tr>
<td>21</td>
<td>Average discharging curves in the idle case study</td>
</tr>
<tr>
<td>22</td>
<td>Monsoon Power Monitor</td>
</tr>
<tr>
<td>23</td>
<td>Samples of Monsoon Power Monitor measurements</td>
</tr>
<tr>
<td>24</td>
<td>Daemon - MPower mean timestamp interval comparison</td>
</tr>
</tbody>
</table>
LIST OF ABBREVIATIONS

3G Third Generation of Mobile Telecommunications Technology. viii, 10, 25

ACPI Advanced Configuration and Power Interface. viii, 19–21, 35, 60

ADB Android Debug Bridge. viii, 61

AIDL Android Interface Definition Language. viii, 63

Android K Android 4.4, codename: KitKat. viii, 8, 42–44, 73

Android L Android 5.0, codename: Lollipop. viii, 6, 8, 11, 38, 41–44

AOSP Android Open Source Project. viii, 6, 72–76, 104

API Application Programming Interface. viii, 2, 5, 7, 9, 10, 12, 17, 23, 24, 29, 32, 33, 43, 46,
47, 51, 54, 55, 99, 102

APK Application Package. viii, 9

APM Advanced Power Management. viii, 19

App application. viii, xii, xiii, 2, 3, 5, 7, 9–12, 14, 15, 18, 25–28, 35, 38, 39, 42, 43, 45–49, 51,
52, 58, 60, 63, 64, 66, 71, 75, 79, 81, 83, 88, 90, 91, 100–102

ART Android Runtime. viii, 8, 42

ASCII American Standard Code for Information Interchange. viii, 56

ASHMEM Android Shared Memory. viii, 63, 64

ASIC Application-Specific Integrated Circuit. viii, 68

BIOS Basic Input-Output System. viii, 19

BLP Battery Lifetime Predictor. viii, 28
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BMU</td>
<td>Battery Monitoring Unit</td>
<td>viii, 23</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
<td>viii, 34, 35, 39, 49, 54, 60, 61, 67, 68, 101, 103</td>
</tr>
<tr>
<td>DAQ</td>
<td>Data AcQuisition</td>
<td>viii, 31</td>
</tr>
<tr>
<td>EDGE</td>
<td>Enhanced Data Rates for GSM Evolution</td>
<td>viii, 10</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field Programmable Gate Array</td>
<td>viii, 103</td>
</tr>
<tr>
<td>FSA</td>
<td>Finite State Automaton</td>
<td>viii, 32</td>
</tr>
<tr>
<td>FSM</td>
<td>Finite State Machine</td>
<td>viii, 34</td>
</tr>
<tr>
<td>GLES</td>
<td>OpenGL for Embedded Systems</td>
<td>viii, 9</td>
</tr>
<tr>
<td>GPS</td>
<td>Global Positioning System</td>
<td>viii, 2, 10, 18, 49, 54, 93, 100</td>
</tr>
<tr>
<td>GSM</td>
<td>Global System for Mobile Communications</td>
<td>viii, 10, 25, 26, 31, 32, 89</td>
</tr>
<tr>
<td>HAL</td>
<td>Hardware Abstraction Layer</td>
<td>viii, 10, 47, 74, 101, 102</td>
</tr>
<tr>
<td>HP</td>
<td>Hewlett Packard</td>
<td>viii, 21, 29</td>
</tr>
<tr>
<td>HPC</td>
<td>Hardware Performance Counter</td>
<td>viii, 33</td>
</tr>
<tr>
<td>HW</td>
<td>hardware</td>
<td>viii, 24, 28</td>
</tr>
<tr>
<td>I/O</td>
<td>Input/Output</td>
<td>viii, 30</td>
</tr>
<tr>
<td>INIT</td>
<td>Initialization</td>
<td>viii, 68, 78</td>
</tr>
<tr>
<td>IPC</td>
<td>Inter-Process Communication</td>
<td>viii, 47, 63</td>
</tr>
<tr>
<td>JNI</td>
<td>Java Native Interface</td>
<td>viii, 9, 60, 65, 74, 81, 82, 102</td>
</tr>
<tr>
<td>JSON</td>
<td>JavaScript Object Notation</td>
<td>viii, 52, 80</td>
</tr>
<tr>
<td>LCD</td>
<td>Liquid Crystal Display</td>
<td>viii, 31, 34, 35</td>
</tr>
<tr>
<td>LTS</td>
<td>Long Term Support</td>
<td>viii, 73</td>
</tr>
</tbody>
</table>
LIST OF ABBREVIATIONS (Continued)

MIT Massachusetts Institute of Technology. viii, 31

NAND Non-Volatile Storage Technology. viii, 71

NDK Android Native Development Kit. viii, 9, 82

NEP Nokia Energy Profiler. viii, 23, 26, 35

OEM Original Equipment Manufacturer. viii, 67–69

OHA Open Handset Alliance. viii, 5

OOM Out of Memory. viii, 63

OpenGL Open Graphics Library. viii, 9

OS Operating System. viii, xii, xiii, 1, 3, 5–8, 10, 12–17, 19, 21, 23, 29–31, 33, 35, 36, 43, 55, 57–64, 66–73, 78, 81, 83–85, 91, 100, 104

OTA over-the-air. viii, 6, 74

PCA Principal Component Analysis. viii, 27

PCI Peripheral Component Interconnect. viii, 31

PDA Personal Digital Assistant. viii, 21, 29

RAM Random Access Memory. viii, 32

ROM Read-Only Memory. viii, 4, 68, 70–73, 79, 101

RPC Remote Procedure Call. viii, 47, 63

SD Secure Digital. viii, 70

SDK Android Software Development Kit. viii, 7, 9, 11, 17, 25, 84

SGL Simple Graphics Library. viii, 9

SPM System Power Management. viii, 19, 60
LIST OF ABBREVIATIONS (Continued)

SSL  Secure Sockets Layer. viii, 9

SW   software. viii, 24, 28

TCP  Transmission Control Protocol. viii, 26

TTC  Time to Charge. viii, 18, 38, 43, 48, 54

TTL  Time to Live. viii, xii, 1, 2, 18, 38, 43, 45, 48, 52–54, 100

UI   User Interface. viii, 15

USB  Universal Serial Bus. viii, 70, 71, 92

VM   Virtual Machine. viii, 8, 64, 73, 74

XML eXtensible Markup Language. viii, 11
SUMMARY

Nowadays, mobile devices are more and more widespread thanks to their flexibility and advanced functionality. They are probably the device we use the most, since they permit us to access to many services (e.g. mail, Internet, social networks, etc.) when we are outside or we do not have another device, like a computer, close at hand. However, if on one hand mobile devices have many advantages, on the other hand, they have drawbacks as well. One of their main drawbacks is certainly power consumption, and this makes smart power management the key point in the enhancement of the user experience. Even though there exist many approaches to the problem, they are often inaccurate in estimating the remaining Time to Live (TTL) of the device. In order to estimate the TTL, it is necessary to gather information about the state of the device. Using an approach based on the device Operating System (OS), we can easily monitor the hardware components that mainly impact on the battery life. In the Android context, there are many applications (Apps) that provide information about the device TTL in this way. However, as far as they are part of the system, they may have a non-negligible impact on the battery life itself. Moreover, in particular cases, it could be necessary to collect such information with a certain frequency, in order to build reasonably efficient and accurate models.

This work proposes an efficient power monitoring approach, deeply fused within the Android OS, able to provide accurate and reliable information about the device status and with a negligible impact on the battery. Most importantly, this work can be applied to contexts where a high
SUMMARY (Continued)

frequency data gathering is required and essential for that particular purpose.

This work is organized as follows:

- in Chapter 1 we define the context of our work and briefly describe its motivations
- in Chapter 2 we provide a general overview of Android OS, present its main features and characteristics and analyze its worldwide diffusion
- Chapter 3 shows the most important works in the mobile devices power management field
- Chapter 4 discusses the problem we want to address and our solution. MPower App is also presented
- in Chapter 5, starting from Android architecture, we explain how our work is implemented, based on the findings in the previous chapter
- Chapter 6 reports the results obtained testing our work from different points of view
- in Chapter 7 we discuss the results together with the limits of our solution and with other researches possibly deriving from this work.
CHAPTER 1

INTRODUCTION

In this chapter, we want to supply an introduction to the power monitoring techniques for mobile devices. In Section 1.1, we present the context of our work, in order to better understand the problem we are tackling and our goals. Then, in Section 1.2, the research field is presented, proposing the literature about power consumption monitoring and estimation, and discussing the proposed approach. Finally, in Section 1.3, we briefly describe our solution and the results we obtained.

1.1 Context definition

In the last decade, the mobile devices market has changed. Indeed, we moved from devices that were mainly able only to make calls and send messages, to the so called smart devices. Such mobile devices (e.g. smartphones and tablets) are considered “smart” thanks to their capacities and flexibility. These devices are becoming more and more popular, and their performance and functionality are growing as well. However, one of the main differences between old and new mobile devices is, for sure, power consumption. Indeed, all these functionality have a non-negligible impact on battery life, which means a reduction in the device Time to Live (TTL). In particular, mobile devices battery life is influenced many by: internal hardware component, user’s behavior and external environment (e.g. signal strength, temperature, etc.).

In a context like Android Operating System (OS), there are many approaches and studies
regarding mobile devices power consumption analysis. Such approaches provide information about the device TTL by mainly analyzing hardware components state. However, since they are part of the system, it is necessary for them to have an impact on battery life as low as possible. Besides, it is important to collect accurate and fresh data about the device state, in order to build a reasonable model of its power consumption. Finally, generalizing this approach, there could be application cases, like the ones related to E-Health, where it is required to gather data at a sufficient high frequency.

1.2 Power monitoring for mobile devices

In order to build efficient power models, it is necessary to monitor the device state and extract the information needed for such purpose. There may be different approaches to this problem. For instance, a possible approach is to use benchmark applications (Apps) on a single device to collect data. However, the models produced in this way hardly generalize to the great amount of devices currently available on the market.

Other approaches aim to exploit users’ behavior to build power models\[5\]. In this way, personalized power models may be built\[6\]. Nonetheless, such approaches do not consider device specific consumption. Therefore, the same user profile may produce different behaviors on different devices.

In this work we focus on the main hardware components that impact the most on battery life (e.g. WiFi, Bluetooth, Global Positioning System (GPS), etc.). Such approaches allow comparison among devices, without the need of controlled environment experiments. Usually, in Android context, data about hardware components are retrieved using the Application Pro-
gramming Interfaces (APIs) provided by the Android Application Framework. In particular, Android System Services are the ones that expose such data to the Application layer. However, it may be not possible to gather all the data required from System Services alone. For instance, screen brightness information are not provided by Android Services. Besides, when we gather such data, the computational overhead necessary to pass through software stack of Android may not be negligible in terms of battery impact. Finally, such overhead may also set an upper bound to data gathering frequency.

In order to overcome such limitations, we propose a low level daemon able to log the device components state.

1.3 Daemon approach motivations

We have seen so far that, in order to retrieve accurate data about the device state, we cannot rely on Android Application Framework. Moreover, such approach may have a non-negligible impact on battery life. Finally, Android software stack may kill or, at least, reduce performance of logging Apps. For this reason, we want to introduce a generic approach for data logging. Our proposal is a low level (i.e. outside of Android Java environment) daemon that gathers information about the device state at a sufficient high frequency. Such data can be easily collected by virtual filesystems: Procfs and Sysfs. In this way, we can enable the usage of Apps that require a logging frequency higher than the one required by power management Apps. Our idea is to create a hierarchical logging system, decoupled from any high level App, but, at the same time, accessible from Apps requiring device state logs.

In order to achieve such result, we want to integrate our daemon as part of Android OS itself.
This can be done using an Android Custom Read-Only Memory (ROM) like CyanogenMod and add our daemon as a native service run as part of the system.

Experimental results have demonstrated that our daemon is able to log the device components state at a sufficiently high frequency (1.5KHz). Besides, there is no statistical evidence to prove that our daemon consumes, on average, more power than the idle state.
CHAPTER 2

BACKGROUND

This chapter presents the background of this work, with a quick review of the Android OS. This review starts from the history of this mobile OS (Section 2.1), then shows briefly some of its features and characteristics (Section 2.2), presents the components of Android Apps (Section 2.3), compares its diffusion with respect to the other mobile OSes (Section 2.4) and, finally, explains why we chose such OS (Section 2.5).

2.1 Android

Android\cite{7} is an open-source software stack for a wide range of mobile devices and a corresponding open-source project led by Google. This stack includes a Linux-based operating system\cite{8}, middleware and key mobile applications and a support set of API libraries to develop Java applications for mobile devices.

The Android project was initially developed in 2003 by Android Inc., founded by Andy Rubin, Rich Miner, Nick Sears and Chris White. In 2005, Google acquired Android Inc.\cite{9} to expand its horizons in the mobile market. In 2007, Google founded the Open Handset Alliance (OHA)\cite{10}, a group of 84 technology and mobile companies who have come together to accelerate innovation in mobile and offer consumers a richer, less expensive, and better mobile experience. Together, they developed Android, the first complete, open, and free mobile platform.

In October 2008, the mobile device HTC Dream (also known as T-Mobile G1) was released.
This mobile device was developed by HTC along with Google and was the first device to run Android OS, in particular Android 1.0 upgradable to 1.6\cite{11}. On November 2014, Google released on the market Android 5.0, codename: Lollipop (Android L)\cite{12}, directly available on Motorola Nexus 6\cite{13} and HTC Nexus 9\cite{14}, and as over-the-air (OTA) update for a selected range of devices. This version includes many interesting power management features, that will be analyzed more in detail in Section 3.4.3. Some statistics on the relative number of devices running a given version of the Android platform are shown in Figure 1 and Table I\cite{15}.

The preferred Android Open Source Project (AOSP) license is Apache Software License, Version 2.0 (Apache 2.0), and the majority of the Android software is licensed with Apache
TABLE I: ANDROID VERSIONS DIFFUSION

<table>
<thead>
<tr>
<th>Version</th>
<th>Codename</th>
<th>API</th>
<th>Distribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.2</td>
<td>Froyo</td>
<td>8</td>
<td>0.6%</td>
</tr>
<tr>
<td>2.3.3 -</td>
<td>Gingerbread</td>
<td>10</td>
<td>9.8%</td>
</tr>
<tr>
<td>2.3.7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.0.3 -</td>
<td>Ice Cream</td>
<td>15</td>
<td>8.5%</td>
</tr>
<tr>
<td>4.0.4 6</td>
<td>Sandwich</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.1.x</td>
<td></td>
<td>16</td>
<td>22.8%</td>
</tr>
<tr>
<td>4.2.x</td>
<td>Jelly Bean</td>
<td>17</td>
<td>20.8%</td>
</tr>
<tr>
<td>4.3</td>
<td></td>
<td>18</td>
<td>7.3%</td>
</tr>
<tr>
<td>4.4</td>
<td>KitKat</td>
<td>19</td>
<td>30.2%</td>
</tr>
</tbody>
</table>

This license promotes openness in mobile world and allows developers, device manufacturers and similar to create, modify and distribute custom version of the software.

2.2 Android features and characteristics

In this section, we want to present the main features and characteristics of Android OS\[1],\[4\] in order to understand and analyze the Android software stack (Figure 2).

Android software stack can be synthesized as a Linux kernel and a collection of C/C++ libraries that are exposed through an application framework that provides services for management of runtime and applications.

All the Apps, native (mail client, browser, calendar, etc), third party and developer, are built on the Application Layer. These Apps are developed in Java programming language and use the same API libraries. The main tool available to develop Android Apps is the Android Software
Development Kit (SDK)\textsuperscript{[17]}. Application layer runs within the Android runtime environment exploiting classes and services made available from the underlying Application Framework. In addition, such framework supplies a generic abstraction for hardware access and manages the user interface and application resources.

Android OS is not simply a mobile Linux implementation and the main reason is the Android runtime. This environment includes the core libraries and the Dalvik Virtual Machine (VM), up to Android 4.4, codename: KitKat (Android K), which was replaced by Android Runtime (ART) from Android L (actually, it coexisted with the Dalvik in Android K). We will focus on Dalvik in Section 5.1.2.3. The core Android C/C++ libraries run on top of the kernel and
provide most of the functionality available in both core Java libraries and Android specific libraries. Some of the common features implemented in these libraries are:

**WebKit engine and Secure Sockets Layer (SSL)** for integrated web browser and Internet security

**Graphics libraries** for 2D and 3D graphics (such as Simple Graphics Library (SGL), OpenGL Graphics Library (OpenGL) and OpenGL for Embedded Systems (GLES))

**SQLite** for native database support

**Media libraries** for playback of a wide range of media formats through *StageFright* custom framework

**Surface Manager** to provide display management

Developers may integrate their Apps with these native libraries and native C/C++ code. The communication among the Java (upper) and C/C++ (lower) level is managed by the Android Native Development Kit (NDK)\[^18\], which, combined with the SDK, generates native code libraries from C and C++ sources and wraps those native libraries in Application Package (APK) files using Java Native Interface (JNI). NDK is mainly used for graphics rendering, sensor input retrieval and legacy code porting. However, it is crucial to understand that the NDK gives access only to a very limited subset of the Android API. Hence, the word “native” in the NDK may be misleading since the use of the NDK still involves all the limitations and requirements that apply to Java app developers. Once the App is ready, the developer has to sign his/her compiled APK files with a certificate he/she owns. The only purpose of this certifi-
cate is to distinguish App authors and it does not need to be signed by a certificate authority. Finally, Linux Kernel handles core services, which include hardware drivers, process and memory management, security, network, and power management, and provides an abstraction layer between the hardware and the remaining of the stack. For instance, the Global System for Mobile Communications (GSM) telephony support is hardware dependent, therefore device manufacturers must provide a Hardware Abstraction Layer (HAL) module in order to enable Android to interface with their hardware. Most of the wireless connection technologies (e.g. Bluetooth, Enhanced Data Rates for GSM Evolution (EDGE), Third Generation of Mobile Telecommunications Technology (3G) and WiFi), are supported by Android and implemented either in Android-specific fashion or in the same way as Linux. Moreover, Android supports sensors like GPS, compass and accelerometer (HAL modules are required for some of them) and makes APIs available in the Application Framework.

So far we presented Android main features, but it is important to stress that maybe the most important feature of Android OS is its open source nature. However, in spite of its licensing, Android is quite different from classic open source projects, which, usually, have public mailing lists, forums where main developers interact with one another and public access to the main development branch’s tip. None of these thing can be found for Android since its development is done mostly behind closed doors. In other words, Android is not a community-driven project. Google (in particular the Android development team) mainly develops Android and the public is not informed about either internal discussions nor the tip of the development branch. For this reason, it is very unlikely that a developer may contribute to Android development. Said
that, Google makes code-drops every time a new version of Android is release (Android L source code was released on November 3, 2014) and its choice of distributing Android under an open source license is a great benefit for developers.

2.3 Android application components

Android Apps consist of loosely tied components which can invoke or use components of other Apps\(^1\)[4]. Moreover, there is no single entry point to an Android App such as main() function or any equivalent. Instead, there are predefined events, called intents, that developers can tie their components to, thereby enabling their components to be activated on the occurrence of the corresponding events. For instance, when a user presses a Contacts button, he/she invokes a component that handles the contacts database. Hence, an App may have many entry points as it has many different components.

The manifest file is probably what is more similar to the main entry point of an App. Basically, it describes each App, specifying its metadata, hardware requirements, SDK platform targeted, software libraries and required permissions. The manifest is formatted as an eXtensible Markup Language (XML) file and resides at the topmost directory of the App sources, as AndroidManifest.xml. All App components, apart from broadcast receivers, which can be registered at runtime, must be statically declared at build time in the manifest file.

Applications building blocks are categorized as follows:

Activities

Activities are the presentation layer of the App and represents the interaction point with the user, showing him/her information through the Views mechanism of interface layouts.
Services

Services are similar to background processes or daemons in the Unix world. Essentially, a service is activated when another component requires its services and typically remains active for the duration required by its caller. Services will be analyzed in Section 5.1.2.4.

Content Providers

Content providers are essentially databases. Usually, an app will include a content provider if it needs to store its data or make them accessible to other apps. All content providers present the same API to Apps, regardless of how they are actually implemented internally. Android C/C++ libraries provides native support for SQLite databases and guarantee high performances, but also files or other types of storage may be used.

Intents

Intents are the Android message-passing mechanism implemented to broadcast messages across the system. They can be used to start or stop activities and services, broadcast information or events regarding the OS or any other user App.

Broadcast Receivers

Broadcast receivers are similar to interrupt handlers. These components are registered to listen for Intents that match a specific criteria, declared using a proper Intent Filter. When a key event occurs, a broadcast receiver is triggered to handle that event on the App behalf.

Notifications

Notifications enable the application to alert the user about particular events, without
2.4 Android worldwide diffusion

The diffusion of Android OS worldwide is here analyzed and compared with the others mobile OSes available on the market (i.e. Apple iOS, Windows Phone, BlackBerry OS).

Smartphone market keeps on growing year after year. Indeed, according to data from the International Data Corporation Worldwide Quarterly Mobile Phone Tracker\textsuperscript{[19]}, the worldwide smartphone market grew 25.3\% in the second quarter of 2014 and established a new quarter record of 301.3 million shipments\textsuperscript{[2]}.

In Figure 3 and Table II, we can see how the worldwide smartphone OS market changed in the last years. On the other side, if we focus on the second quarter of 2014, Android continues...
TABLE II: WORLDWIDE MOBILE OPERATING SYSTEMS DIFFUSION

<table>
<thead>
<tr>
<th>Period</th>
<th>Android</th>
<th>iOS</th>
<th>Windows Phone</th>
<th>BlackBerry OS</th>
<th>Others</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q2 2014</td>
<td>84.7%</td>
<td>11.7%</td>
<td>2.5%</td>
<td>0.5%</td>
<td>0.7%</td>
</tr>
<tr>
<td>Q2 2013</td>
<td>79.6%</td>
<td>13.0%</td>
<td>3.4%</td>
<td>2.8%</td>
<td>1.2%</td>
</tr>
<tr>
<td>Q2 2012</td>
<td>69.3%</td>
<td>16.6%</td>
<td>3.1%</td>
<td>4.9%</td>
<td>6.1%</td>
</tr>
<tr>
<td>Q2 2011</td>
<td>36.1%</td>
<td>18.3%</td>
<td>1.2%</td>
<td>13.6%</td>
<td>30.8%</td>
</tr>
</tbody>
</table>

to dominate the global smartphone market, with almost 85% of the market share in the second quarter of 2014. For sure, one of the reason of this result is the accessible price of most Android-based devices. The largest vendors of Android-based devices are Samsung, Huawei, Lenovo and LG. On the other side, as non-Android products, we mainly find Apple iOS-based devices, sharing nearly 12% of the market and Windows Phone at 2.5%. BlackBerry OS and other mobile OSes cover less than 1% of the market.

A very important part of the smartphone market is certainly composed by App Stores. The top three App Stores are: Google Play Store, Apple App Store and Windows Phone Store. Until a few years ago, Apple App Store was the leading one in terms of Apps available. On October 2012\[^{20}\], Play Store reached 700,000 Apps available for downloading onto Android-based mobile devices and matched the number of Apps available on Apple App Store. Then, on January 2013\[^{21}\], Google Play store overtook Apple App Store; indeed, 800,000 Apps were available on Google Play, whereas Apple App Store contained 775,000 Apps and Windows
Phone Store offered 150,000 Apps. Figure 4 shows the number of Apps available in leading App stores as of July 2014.

2.5 Android choice

It is fairly easy to understand why Android has been chosen as the target mobile OS for this work: it is the most widespread open source OS for mobile devices, owns almost 85% of the market and has the greatest choice of Apps available in its store, as shown in Section 2.4. There are other open source OSes, like Samsung Bada\(^2\) and Symbian\(^3\), that are no more maintained and their presence on the market is extremely restricted. Another alternative could be Sailfish\(^4\), an open source OS by Jolla\(^5\) based on Linux, Mer\(^6\) and Qt\(^7\). Many Android applications run on Jolla devices unchanged. Moreover, if a developer wants to take advantage of all User Interface (UI) and other features of Sailfish OS, he/she can port the App to native
Qt/QML. However, at the moment, Sailfish OS is not so spread and, for this reason, it has been put aside for this work.

Android, thanks to its widespread use, has a strong community that contributes to the project and maintains its documentation. Its structure and its source code are available and can be easily downloaded and built\cite{28}.

In this work, we are interested in analyzing mobile devices power consumption. However, Android does not provide instruments to understand how much the whole system is consuming. Starting from version 2.3 (Gingerbread), Android has added some statistics for the final user, i.e., a battery usage report that can be accessed through the system settings menu (Figure 5).
This screen lists all the applications and services that have been consuming at least the 2% of the battery life, since last full battery charge, displays a graph showing the battery discharging over time and reports some useful data (for each listed application). However, the main drawback is that these information are on the past behavior of the device and no prediction on the future battery discharge trend is made. Moreover, Android SDK includes a set of APIs useful to access the available sensors and observe the device context. This is not the case of other proprietary mobile OSes like iOS, BlackBerry or Windows Phone 8, where the same information cannot be directly accessed in such an easy way. Therefore, Android, thanks to its widespread use and the large amount of information that can be directly retrieved, is the best choice for monitoring power consumption.
CHAPTER 3

STATE OF THE ART

This chapter presents the works that contributes to the state of the art in the field, with an emphasis on works about mobile power monitoring and management. The objective of the techniques presented here is to collect information about mobile device status to generate a power model of the device itself. A power model is a mathematical model in charge of predicting the future power state of a system. In mobile devices context, the power model analyzes the device discharge curves, and identifies the battery behavior in particular condition (e.g. WiFi on, GPS on, etc.). In this way, the power model can provide predictions about the device TTL and Time to Charge (TTC). TTL may be defined as the time window necessary to completely discharge the device. On the other hand, TTC is the time window necessary to completely charge the device.

In Section 3.1 we present the problem of power management in computers, in particular on Linux machines. Then, we will focus on mobile devices and show a survey about the main methodologies for power consumption analysis available in literature (Section 3.2). Finally, we will present the most famous and used Apps that perform power consumption analysis and optimization on Android (Section 3.4).
3.1 **Power management in computers**

Power management has different purposes as far as we are considering desktop or laptop computers or mobile devices. On one hand, power consumption reduction in standard computers will lower the amount of energy used for heat dissipation in favor of saving money and increasing the stability and durability of the system components. On the other hand, power management in mobile devices is particularly critical: the energy available is limited and the system will shut down as soon as it is over.

Nowadays, nearly all laptops include built-in battery instrumentation that the OS inspects to read capacity, to calculate drain and charge rates, and to receive capacity alarms. Moreover, the OS can provide multiple policies of power saving, which include, for instance, disabling the memory cache, applying clock gating, voltage scaling, activating sleep modes and so on. These features are managed by the Advanced Power Management (APM)\[^{29}\] or the Advanced Configuration and Power Interface (ACPI)\[^{30}\]. APM is a simple, Basic Input-Output System (BIOS) based power management system, still used on old computers. On the other side, the ACPI approach allows power management control from within the operating system. In this way, priority-based policies can be declared in order to turn on only needed hardware peripherals and suspend non-critical applications that, at the moment, are no longer active. System behavior can be customize by both the user and the OS to match their needs. Besides, Linux implements a functionality called System Power Management (SPM), which can place the entire system in a low power state. In this state, the system is still able to guarantee a low response latency when the user asks for OS functionality, but the overall power consumption is reduced
to its minimal amount with respect to its standard operating condition.

However, power consumption management and analysis on computer is not so easy and immediate. On Linux, the content of: `/proc/acpi/battery/*` exposes information and state of the battery, although, sometimes, such data are not very accurate and may be misleading. In [31], the authors plot the battery capacity from fully charged until depleted in order to find out if the drain rate suggested by the ACPI was accurate. If the system is running a constant workload, such as the idle one, then the instrumentation should report: full capacity equal to the design capacity of the battery at the start, 0 capacity just as the lights go out and a straight line in between. However, only new and properly conditioned batteries do this whereas old and not conditioned batteries tend to supply very poor capacity data.

Usually, power characterization in computing platforms is typically obtained through either direct measurements (by using physical instrumentation) or modeling based on hardware performance counters. In particular, modeling techniques based on linear regression are commonly used. However, in [32] the authors showed that these techniques frequently exhibit high prediction error in modern computing platforms. Indeed, their inherent complexities, such as multiple cores, require new modeling techniques in order to challenge the rising hardware complexity and variability. In particular, the authors stated, thanks to a comprehensive measurement framework and an extensive set of benchmarks, that prediction of subsystems power consumption based on linear regression models, often perform poorly (10-14% mean relative error, 150% worse case error for the CPU) and more complex non-linear models do only marginally better. Hence, in authors’ opinion, pervasive, low-cost ways of measuring instantaneous subsystem
In [33], a novel, history-based, statistical technique for online battery lifetime prediction is presented. The idea of this approach is to take a one-time, full cycle, voltage measurement of a constant load and use it to apply a coordinate transformation that converts a dynamic voltage curve into a form with robust predictability. Then, a statistical method is applied to make a flexible lifetime prediction based on the transformed history curve. The authors implemented this method on Hewlett Packard (HP) iPAQ Personal Digital Assistant (PDA) [34] running Linux and compared its results to those provided by two native Linux battery lifetime prediction systems: the ACPI and Smart Battery [35]. They both implement fast, on-line, portable prediction method at the OS level, in particular ACPI uses the division of remaining battery capacity and present rate of battery drain to estimate remaining battery life. On the other hand, Smart Battery uses a rolling average algorithm that, at each prediction point, first calculates the average current within last one minute; then takes the division of present remaining battery capacity and the average current as the present battery life estimation. However, the authors stated that such simple approaches to prediction consider only a very short discharge history and thus can be highly inaccurate, while their method is more efficient, accurate and easily adaptable to different batteries and systems.

3.2 Power models for mobile devices

Smartphones are devices that feature both old mobile phones and modern laptop computers characteristics. On one hand they are small and light, while on the other hand they offer computer-like user experience and capabilities. As result, we have a completely new kind
of devices which are mostly used like computers, but, unfortunately, with strong power and availability constraints. The main factors impacting battery life are explained here, alongside the most common methods of obtaining data about power consumption.

3.2.1 Battery draining factors

Smartphones are complex systems: it may not be so straightforward to collect the necessary information to build a precise characterization of their power consumption. However, we can trace their power model back to three main contributions.

**Hardware components** are the basic components of each smart device. They are the first choice when trying to build a power model, because they can be easily studied in laboratory conditions, although it is not trivial to grasp all the internal dependencies between them have when used in real-world conditions.

**User’s behavior** has a great impact on devices power consumption[5]. Moreover, it is possible to create a great variety of discharge profiles since almost each person has a different usage pattern.

**Environmental conditions** influence devices discharge and must be taken into account. Indeed, due to environmental conditions (e.g. signal strength, temperature, etc.), two different power models may be generate for two users even though they own the same device and present the same usage pattern.
3.2.2 Approaches to the problem

The first issue to overcome, in order to build a power model consists in getting data to work with. With respect to laptops, the issues to face when collecting data are only slightly different, as we discussed in Section 3.1. Indeed, the data gathering phase must pay particular attention not to waste the very limited battery life. There are three main approaches to this problem.

Direct measurements in laboratory

Direct measurements require to dismantle the device and measure the power drain of every component. Although it is the fastest and most precise way of measuring the consumption given by the devices components, it is also the less flexible, because, in order to build a general model, it requires to have every phone available on the market to perform the analysis in laboratory.

On-phone special instrumentation

Some devices come equipped with dedicated hardware, useful for conducting power-related studies. Nokia Energy Profiler (NEP) and the Battery Monitoring Unit (BMU)\cite{36} are examples of such special instrumentation. These systems are a good trade off between versatility and precision, but, they are available only on some devices.

OS APIs

On Android, it is possible to gather information about the component state and battery status by just using the APIs. In this way, it would be possible to virtually build a model for any Android device. However, those APIs are not very precise and, often, change with new Android versions, making difficult to support them for data gathering.
3.3 Power consumption measurement methodologies

In this section, we present a survey on the main power consumption methodologies for mobile devices available in literature. We will analyze and compare the models proposed by the works in the field in order to extract useful information about which smartphone components have to be monitored to generate a good power model.

In order to compare and analyze the different research works regarding power consumption monitoring and modeling on mobile devices, we have to define a set of categories, to group similar methodologies and evidence their differences. The three classifications we propose here are an high level view of existing methodologies based on the level used to retrieve data, which induces a different approach in modeling.

**Application level**

Here, the power consumption modeling is performed per single application or per applications with respect to their specific workload characteristics.

**User level**

The user behavior and the impact of different usage patterns in overall power consumption are considered in this category.

**System level**

This category goal is to develop a model able to compute the whole system power consumption at a given time. Power measurements may be performed by using external systems, internal hardware (HW)/software (SW) components or via internal APIs.
Clearly, a proper description of each method features is required to have a more detailed analysis of the embedded hardware components considered to build the models.

### 3.3.1 Application level

Here, we want to present works whose goal is to profile application power consumption. They perform a more fine-grained profiling, compared to system-wide models and, some of them, also study the impact of network data transmission on the single Apps; indeed, network activity consumes a significant part of the available battery power.

*SPOT*, a tool creating power-aware applications, is proposed in\(^\text{[37]}\). The main idea is to predict the power consumption of an application during its development identifying hot-spots inside the code before the application is released and tested on an actual device. SPOT, at first, allows developers to create high-level architectures representing a program and the used components, then, it creates the code and integrates the information about the power consumption, with an error of 3-4%, with respect to actual power consumption. SPOT can also, alternatively, create the code to be run on a real device; in this case, some logging functionality are added in order to save useful information to refine the power model offline. However, the main drawbacks of this tool are the lack of integration with the SDK and the limited number of devices available to test an application.

In\(^\text{[38]}\), mobile network available technology (i.e. 3G, GSM and WiFi) is considered in order to model the energy consumption of Apps requiring a network activity. In particular, this study considers, as main power consumption contributions, the transmission energy and the Radio Resource Control protocol, responsible for scaling the power consumed by the radio based on
inactivity timers. NEP was used to gather precise power consumption information in real-time, while, on not Nokia devices, measurements were taken using an external hardware power meter. Results showed that GSM consumes 40% to 70% less energy compared to 3G to download data, but WiFi is more energy efficient than both cellular networks once it is associated. The energy model was build taking into account both the size of transfer and the time between consecutive transfers.

On the other hand, the analysis performed in [39] is focused only on the WiFi connection. NEP and external measurement system were used also in this case. During the measurements, the basic components of the devices were in use, to minimize the dependency with other components. The authors collected data on power consumption of the testing Apps having the network interface in different operational modes and during Transmission Control Protocol (TCP) download at different data rates. They modeled the power consumption using a set of linear equations and estimated the power of a download and the power consumed by an upload with a mean absolute percentage error of 6.8% and 5.8%, respectively.

In [40], application power profiling is used for malware detection. The authors calculated the power consumption of various activities done by the user (e.g. calling or surfing the Web) and, using these measures as base level, they monitored power consumption during use, in order to spot abnormal consumption. Then, during phone recharge phase, they performed a more fine-grained analysis on each App to spot the ones that are contributing to the power consumption variations. Thanks to this analysis, they managed to achieve a 89% detection rate, in case of a message forwarding malware.
3.3.2 User level

Each user has different needs and device usages, hence, it would be reductive to build efficient power consumption models without taking into consideration the impact of the user. Indeed, mobile devices are able to run several different Apps and the amount of time spent by a user on each one of those contributes to determine the device battery lifetime. We now present works where the user experience is taken into account in order to build a proper power consumption model.

In [5], it is explained why models ignoring the users usage patterns are limited. The authors developed an application collecting data about the users; then, they applied the Principal Component Analysis (PCA), a statistical technique used to discover the component with the biggest impact on power consumption. As result, the authors showed that the most consuming components where strictly dependent on the user, since different users have different principal components. In the case we do not consider users modeling, powerful power optimization may not be operated on unused components. However, this work showed that PCA can be used to highlight each device major consuming components, in order to build personalized power models.

In [41], a power model considering the usage pattern is created. In this way, the limitation of models considering only the device may be overcome. The authors assumed that a mobile device has $n$ different states that may be represented as a tuple containing the condition of the phone components. For instance, a possible state is $(\text{LCD};\text{VOICE};\text{DATA}) = (\text{ON};\text{ON};\text{ON})$. As result, the authors noticed that users spend different time into each state. This study advanced
in[6], where the authors tried to create a daily and weekly profile reflecting life patterns (e.g. working, sleeping, etc.). In order to measure the battery consumption and time spent in each state, they used an application that, periodically, records the information about usage patterns and power consumption to a log file, which is sent and processed by a server, to build the model. This work patent was published on 2012[42].

The study presented in[43] concerned how intentional user activities impact on the device power consumption. It is claimed that the energy consumption is influenced by platform (HW/SW) and Apps/user interactions and, as result, user activities contribute heavily towards energy drain. In[44], the same authors proposed a study about network traffic on smartphones. They implemented a tool that provided an application-level view of smartphone traffic and analyzed the data produced. They found out that, on one hand browsing contributes over 50% of the traffic (email, media and maps contribute to roughly 10% each), on the other most data transfer are small, making the radio controller waste power in its sleep-active-idle cycle. A reduction of tail timed from 12 seconds to 4.5 allows to save 35% power in radio communications. Moreover, a perfect knowledge of the incoming traffic permits to save up to 60% power, meaning that this methodology can be improved by a model considering usage patterns.

One of the main components in[45] is the Battery Lifetime Predictor (BLP). The used approach is to compare the device actual discharge with a measured base curve, obtained when the device was in idle. In order to compute the base curve, it is necessary a periodically one-time offline measurement, to consider also battery aging phenomena. However, this method works only with a predefined set of measured applications; hence, the prediction will not be valid anymore
if a new application is added. CABMAN was prototyped for both Linux and Symbian OS, using as test devices a HP laptop with a new battery, a Dell laptop with a very old battery and a regular HP iPAQ PDA. The battery lifetime predictor has a percentage error rate varying between 1.2%, having a new battery, and 6.1%, when the battery is very old.

3.3.3 System level

A large number of works aim at creating power models of the entire smart device. Data to build system-wide power models can be gathered in two ways; either by attaching external sensors to the device (offline method), or relying on the device APIs (online method). The former methodology produces an accurate inspection of the mobile device using predetermined and controlled conditions. Currently, a great variety of mobile devices exists and it is well-known that their behavior is affected by the OS release version, thus it is almost unfeasible to generate offline power consumption model for each combination of mobile device and OS version. Moreover, this analysis does not evolve with the running life of the device, while a runtime generated model is able to adapt itself to new software updates or even new devices. On the other hand, an adaptive model generation relying only on software APIs to gather information on the actual battery state cannot be as precise as the one developed using external measurement system. Some smartphones are given a wider set of hardware sensors, hence allowing an intermediate approach to be used, with internal sensors providing precise data about the device power consumption. This last approach provides intermediate precision results between the external and the API approach, but it lacks of flexibility, since it can be applied only to those devices having internal sensors.
3.3.3.1 **External measurements**

External tools may gather very precise data from the device, however it is difficult to apply this approach, given its invasive nature on the device.

The goal in\(^{[46]}\) is to decompose the energy consumption into independent components. The power consumption is measured by inserting a resistor in series between the battery and its connector on the phone. A sampling board is used to measure the battery voltage. Measurements are centrally managed by a *Power Server* and do not require user interaction. Tests were specifically designed to stress each component considered and to provide the model for battery consumption addressed by the specific component. The analysis was computed by sending data over a wireless network, using an Android device. Collected traces allowed to analyze the energy consumption involved in many different communication phases.

The study in\(^{[47]}\) by University of Michigan proposed a 2-way OS-level power management. This system consists in a modified version of Linux, running on a handheld iPAQ. Input/Output (I/O) peripherals are fully described using a state enumeration, called *power modes*, each one associated to a power consumption values expressed in \text{mW}. To measure the power states, the authors removed the battery from the iPAQ device and sampled current drawn at 50Hz through the external power supply. In this system, applications are allowed to query the power mode of I/O peripherals, and, moreover, they can disclose hints about expected device discharge rate in a different power-mode. The system provides also a decision phase for power saving, relying on heuristics. Experiments are performed focusing on a web browser application and an email reader application.
Cinder\textsuperscript{[48]} is an OS based on HiStar kernel\textsuperscript{[49]}, developed by Stanford University and Massachusetts Institute of Technology (MIT) CSAIL. Cinder provides low-level abstraction for energy management and it is specifically meant for mobile, energy constrained devices. This research work does not directly focus on the power modeling, however it is interesting that they clearly stated that Cinder needs a solid model in order to work properly.

In\textsuperscript{[50]} a very deep analysis of power consumption on smartphones was made. The goal was to understand where and how energy is used, in order to provide basis for understanding and managing mobile device energy consumption. They profiled energy consumption taking physical power measurements of supply voltage and current at the component level on a piece of real hardware. They inserted sense resistors on the power supply rails of the relevant components to measure current. They also used a National Instruments Peripheral Component Interconnect (PCI)-6229 Data AcQuisition (DAQ)\textsuperscript{[51]}, connected the sense resistors via twisted-pair wiring, to measure voltages. Using this kind of measurement the authors were able to directly measure the power consumed by the main components of the mobile device. Furthermore, they measured total power consumption by inserting a sense resistor between the power supply and the device. In order to construct their model they ran two different benchmarks: at first, a series of micro-benchmarks designed to independently characterize single components, then a series of macro-benchmarks based on real usage scenarios. From those studies emerged that the majority of power consumption can be attributed to the GSM module, the display (including Liquid Crystal Display (LCD) panel and touchscreen), the graphics accelerator/driver, and the backlight.
In almost all benchmarks, the brightness of the backlight is the most critical factor in determining power consumption. They also showed the impact of the screen image on power consumption. The GSM module consumes a great deal of both static and dynamic power (both maintaining a connection with the network and during a phone call), while the Random Access Memory (RAM), audio and flash subsystems consistently showed the lowest power consumption. Thanks to the data collected, the authors built an energy model based on usage patterns in order to understand when the major part of energy is wasted: for this reason, they defined a set of patterns (e.g. suspend, regular, business) of the smartphone usage and described the daily energy use and battery life under each of them. This kind of analysis clearly showed how different kind of usage have different impact on the power consumption.

The work described in [52] presents both a power-modeling approach for Android-based mobile devices and a novel energy-saving policy to manage the screen brightness. Measurements are performed with a Fluke i30 AC/DC current clamp [53], while the operating voltage is retrieved through Android APIs. The modeling phase consists of a two states Finite State Automaton (FSA): stand-by mode and active mode. If in the former state the consumption is accounted as fixed, in the latter the power consumption is computed through a linear regression model, given by the R-Tool fed with data collected by stressing a single hardware components considered. This models can predict the power behavior of each scenario with a median relative error of 6.6%. Moreover, the authors used data coming from several users to identify the most consuming component (the screen). By making the system automatically adjust the screen brightness, they managed to save up to 10% total energy, with minimal impact on user satisfaction.
3.3.3.2 Only APIs

Different approaches rely only on the device APIs to collect the necessary data to build the power models. Although OS APIs have some limitations in terms of precision, they are present on every device, thus allowing these power models to be more adaptable.

In \cite{54}, the authors present a methodology for building a system-level power model, without requiring laboratory measurements. This model relies only on the data provided by the OS, hence it can be exported to any device. They developed a linear regression model with non-negative coefficients, describing the aggregate power consumption of the processor, the wireless network interface and the display. They analyzed data on the activity levels of each hardware component, such as the Hardware Performance Counters (HPCs) for processor, the downlink and uplink data rates for the wireless interfaces, and the brightness level for the display. They discovered a linear regression model is sufficient to model the relation between the variables they chose and the power consumption. To construct the model they used five types of different workloads: idle with different brightness levels, audio/video players, audio/video recorders, file download/upload at different network data rates, and data streaming. An interesting feature of this model is the fact that it can be adapted to new hardware. In fact, when a new hardware component is installed into the mobile device, it is possible to add regression variables, describing the activity levels of the new hardware component, defining new test cases to stress the new components and, finally, fitting the new data sets to a regression model. In the case an analytic power model of the new hardware component is provided, it is possible to merge it with the existing system-level model. The power estimation, based on this model, exhibits a
median error of 2.62% in real mobile internet services. Moreover, they provide a power model that is independent from usage scenarios and that can be used for runtime power estimation with reasonable accuracy.

The study in [55] proposes a context-aware system to accurately predict the battery lifetime. The energy consumption of each system component is considered dependent on its operational state and on the amount of time it remains in that state. Therefore, the system power consumption is modeled as the sum of the system components. Data about the discharge rate were collected in several system contexts, where a single system context is a combination of factors such as Central Processing Unit (CPU) usage, LCD brightness, WiFi state. The system was tested using a T-Mobile G1 smartphone [11], running Android. Data were collected using 40 different test scenarios: 16 were used to build the model, the remaining to evaluate the generated model. The model generated predict the battery remaining lifetime with a relative error of

The power modeling approach in [56] is based on tracing system calls of the applications that capture both usage-based and non-usage-based power behavior. It is based on a fine-grained energy estimation. The scheme consists of two major components: a Finite State Machine (FSM) (to model the power states and state transitions) and a linear regression model. Some of the FSM states have constant power consumption, to describe non-usage power consumption, while other states leverage a linear regression to capture the power consumption due to system calls generating workload. Moreover, the FSM transition rules are systematically uncovered by a testing application. This new modeling approach improves the accuracy of fine-grained energy estimation compared to usage-based model. Indeed, their model have a 80th percentile
error of less than 10% estimating the power consumption of 50ms of a generic App execution, while the usage-based model has an error that varies between 16% and 52%. Its error for the whole App, with 1 second granularity, varies between 0.2% and 3.6%, compared to the error 3.5-20.3% given by usage-based model.

3.3.3.3 Custom and Internal Measurements

As we have seen, several works have been specifically designed to exploit interfaces already available on specific devices or OS, such as the ACPI or the NEP. These interfaces allow to gather precise information about the battery status (like voltage, current, temperature, etc.) or about the current power consumption of the device at a specific time. In this way, it is possible to generate very precise power consumption model. However these models are not portable on devices without specific interfaces; in fact, some Android devices may be unable to provide the required data with the required precision, hence they cannot take advantage of these power models.

In [57], Sesame system is presented. It is a self-modeling approach to build high-rate mobile system models without any need for external measurement systems. In the beginning, Sesame collects data traces (system statistics and data provided by the ACPI), and builds an initial set of predictors; they may be user defined and depend on the platform in use. Predictors may include factors such as CPU usage, cache misses, WiFi traffic, LCD backlight level and so on. Then, the model is built using two iterative techniques, model molding and predictor transformation. The former generates a model using linear regression, while the latter improves the accuracy of a molded model, transforming the original predictors and finding better linear
combinations of the original predictors. Sesame was implemented both for laptops and smartphones, using a Linux kernel. The system overhead, in the worst case, has been computed to be 3% and 12% on the laptop and on the smartphone, respectively, while typical values are 1% and 5%, respectively. This work showed that data collection frequency influences the system accuracy, and it drops drastically when the data collection frequency matches the frequency of the register used to store the monitored value. Accuracy, on the smartphone, was reported as 86% and 82% at 1 Hz and 100 Hz, while, on the laptop, 95% and 88% at 1 Hz and 100 Hz.

In [58], an online generation of power models is proposed. The authors started by analyzing, with external tools, each component separately. They concluded the power consumption of major components affects the system independently, hence, power consumption of the entire system is just the sum of the contribution of the active components. They also noticed that different devices have power model pretty far from each other, therefore a power model for each device is required. They modified their method to be able to create a power model, based on data coming from sensors within each device. The result indicates that the power model built with PowerBooter is accurate to within 4.1% of measured values for 10-second intervals. Nonetheless, this model presents two main limitations: the need of a specific discharge curve for every specific device and the need for a mobile phone that allows superuser access in the OS.

3.3.4 Results of the survey

Here we present a summary of the methodologies we compared so far. Generally, offline data gathering gives more precise results, but is more difficult to implement and consequently less prone to adaptation. As a consequence, they are rarely used to give the users an estimation of
TABLE III: COMPONENTS CONSIDERED BY PAPERS IN THIS SURVEY

<table>
<thead>
<tr>
<th>Work</th>
<th>HW Components Treated as Independent</th>
<th>Screen</th>
<th>CPU</th>
<th>GPS</th>
<th>SD Card</th>
<th>Bluetooth</th>
<th>Audio</th>
<th>Network</th>
<th>WiFi</th>
<th>None</th>
</tr>
</thead>
<tbody>
<tr>
<td>[52]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>[47]</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>[54]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>[57]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>[56]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>[55]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>[5]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>[6]</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

lifetime battery. Besides, this methodology lacks of generality. On the other hand, it is possible to perform an online data gathering and to build power model at runtime. Although these models are not as precise as offline models, they have a great adaptability and are the most suitable to cope with the fast growing market of smart devices. Moreover, it is important to separate models taking into account the user interaction from the ones which do not. The former models provide a better estimation of lifetime battery, since they model the behavior of every single user, while the latter models are built using benchmark applications that are built to stress one component at the time.

Said that, this survey was useful to have a clear idea of which are the main components that influence power consumption the most in order to focus our work on those components. Table III shows the components analyzed by some relevant papers of the survey.
3.4 Power consumption Apps

Here we present two remarkable and spread Android Apps able to predict the TTL and TTC of the battery, to optimize its usage and to suggest device configuration to the user, in order to increase battery TTL. Of course, we will just give a general overview of such Apps, since information about the power model computation are not available. Finally, we will analyze the new features available in Android L, regarding of battery saving and TTL prediction.

3.4.1 Snapdragon™ BatteryGuru

Snapdragon™ BatteryGuru is a well-known App (Figure 6) entirely developed by Qualcomm Connected Experiences, Inc.[59]. The main goal that supported the whole design and optimization of this app is to enhance battery performance, as the exotic name let the public easily get, by reducing power consumption via a carefully crafted optimization of the device functionality, only on devices featuring Snapdragon mobile processors. According to Google PlayStore[60], this App has been downloaded by millions of users, and evaluated with 4.1 stars over 5. In 2-4 days, it analyzes the user’s behavior, learns how the smartphone is used and starts to optimize the battery usage. For sure, one of the key points is the fact that this App is made by the processor manufacturer. Therefore, the power model the App generates is designed to work well on those processors and this is certainly a great advantage.

3.4.2 Battery Doctor

Battery Doctor[61] is another famous App for power management (Figure 7). It is developed by Cheetah Mobile Inc. and has been downloaded by more than 150 million users on both for Android and iOS[62]. This App, apart from providing the TTL and TTC of the device
and optimizing battery life, implements a *Task Killer* that closes all the Apps running at that moment. Moreover, it provides (inside the App itself) the toggles for Bluetooth, data connection and WiFi, and allows the user to control brightness. If the device has root access, the user may also manage the CPU performance.

Battery Doctor has a interesting unique 3 stage charging system that regulates electric flow to ensure that device charges quickly and fully. These stages are:

**Fast Charge**

Recharge is not limited in Fast Charge phase, in order to let the battery capacity reach 80% as fast as possible.
Continuous Charge

Recharge proceeds regularly up to complete charge.

Trickle Charge

This phase controls charge, so that it cannot reach 100% and, consequently, avoid overcharge phenomenon.

Finally, Battery Doctor gives the user the possibility to choose and activate energy-saving profiles, both predefined and customized by the user. Each profile handles device features such as brightness, screen time-out, connectivity (Bluetooth, data connection, WiFi), ringer volume and vibration. The profiles can be activated either manually or automatically; in this case, the
user can schedule activation or deactivation of a certain profile in a particular moment of the
day or week.

3.4.3 Android L

As said in Section 2.1, Android L was released on November 2014\textsuperscript{[12]}. Certainly, one of
the most anticipated features of Android L was \textit{Project Volta}. This name represents a suite of
enhancements introduced by Google to boost smartphone battery life.

This suite is mainly composed by four components:

Battery Historian

Google created this tool\textsuperscript{[63]} to allow developers to visualize exactly what is using smart-
Figure 8: Battery historian

phone battery, to what extent, and at what time (Figure 8). In this way, developers can profile the energy consumed by their Apps.

the ART runtime

ART is the new runtime environment introduced in Android L to substitute Dalvik. ART was already available in Android K where it coexisted with Dalvik, but this version is definitely more advanced. ART appears to be more efficient than Dalvik, since, while the latter compiles code every time an App is run (just-in-time compiling), the former compiles and optimizes the App source code only once. Therefore, the processor spends less time in compiling and consumes less power. Moreover, there are many technical improvements (e.g. improved garbage collector) that improve efficiency and battery life.
Battery Saver mode

While, on Android K, the OS reported only battery usage information on the past behavior of the device, Android L predicts the TTL of the battery (Figure 8a) and, if the smartphone is in charge, the TTC. We suppose that Android L features the results presented in\cite{58}, since that work was in part supported by Google itself. In addition, Google introduced a feature called Battery Saver (Figure 8b) that, if the battery drops beneath a certain threshold (15% by default), reduces power consumption by, for instance, lowering processor speed, reducing animations and so on. The idea is to allow the user to get to the next charge without shutting down the smartphone.

JobScheduler API

JobScheduler API allows developers to define specific background tasks that should happen when certain conditions are met. For instance, developers may define that their App may send data to their server only when the smartphone is connected to WiFi and plugged in. Previously, developers were able to implement such rule-based system, but it was necessary to wake the device up to determine its state. Now, JobScheduler API processes job requests from various apps in background, and, when conditions are met, the system batches together all the necessary tasks, and performs them at once.

Ars Technica\cite{64} compared power consumption on a Nexus 5 running both Android K and the developer preview of Android L. The battery test involved: a fully charged smartphone (Battery Saver disabled on Android L), brightness set 200 nits, a script to refresh the same set of web pages over WiFi every 15 seconds. Battery on Android K lasted 345 minutes, while on the
developer preview of Android L it lasted 471 minutes (almost 36% longer than Android K). Although the test was performed in a developer preview, the results seemed to be encouraging.
CHAPTER 4

PROBLEM DEFINITION AND PROPOSED SOLUTION

According to what we addressed so far, it is clear that our work will focus on power management in mobile devices. In particular, we will focus on power monitoring. We want to monitor the device components in order to extract useful information that may help us to compute an efficient and accurate power model. We have seen in Chapter 3 that different studies have approached such problem and tried to build system able to suggest the user the status of his/her device battery and tips about its management. It is not our purpose to build a such system from scratches. In fact, we will present, as case study, the App MPower and enhance its device state monitoring system. First of all, we will define the problem we are addressing (Section 4.1). Then, we will present MPower App and explain how it works in details (Section 4.2). Finally, our proposed solution will be revealed (Section 4.3).

4.1 Problem definition

So far, we have seen that one of the main topics in the context of mobile devices is power management. However, even though there exists many approaches to the problem, they are often inaccurate in estimating the remaining TTL of the device. Moreover, as far as they are part of the system, they may have a non-negligible impact on the battery life itself. Finally, in particular cases, it could be necessary to collect such information with a certain frequency, in order to build reasonably efficient and accurate models. Consequently, power consumption
(usually supposed to be a non-functional requirement) turns into a functional constraint, in particular in such Apps. Therefore, their design gets more and more complex.

Inside the Application Framework level, we can use many of the features and APIs provided by such framework. In particular, many information about the state of the device can be retrieved by Android Services. For instance, if we want to retrieve information about the audio, there is an API that calls the *audio service manager*, which returns information such as current output speaker level and so on. Android has abstraction layers, and, inside the
Android App, when we call the audio service, we do not care either about the audio driver nor about who provides such information. From Application level, we are making an Inter-Process Communication (IPC) call to the lower native services (Figure 9). The mechanism in charge of handling IPC and Remote Procedure Call (RPC) is called *Binder* (we will examine it in Section 5.1.2.2). So, when we call the audio service manager from the Application level, we are actually calling some lower native service. In Android, the HAL wraps all the drivers. Such drivers must be written according to specific pattern, so that, at runtime, the HAL knows what are the registered driver, and knows how to call the driver APIs for some functionality. It is clear that traversing this stratified API stack complicates and slow down the logging process.

In this prospective, the App **MPower** is our case study and we want to enhance its logging service. Indeed, at the moment, MPower logging service relies on Android services. However, not all the information needed by MPower to compute the power models can be retrieved in such way, since some of them are inaccurate (e.g. screen service). For such information, MPower relies on other Android APIs that may not be as efficient as Android Services. Moreover, collecting information from Android services may not have a negligible impact on the battery. Finally, tests have proved that MPower logging service cannot go further than a certain logging limit. Indeed, even though we set MPower logging service to log the device state every millisecond, MPower, probably due to the callbacks needed to collect the device state, executes a log almost each second. We ran MPower in such configuration for a while and extracted 1000 timestamps from its database. We computed the difference between each timestamp and calculated the mean value. It turned out that, on the average, MPower logged device state every 1.25 seconds.
(0.8Hz), with a standard deviation of 427 milliseconds. Therefore, this is another limitation we want to overtake.

We now want to give an overview of MPower App infrastructure.

4.2 MPower

MPower is the result of a research project at NECSTLab\textsuperscript{[65]} (Politecnico di Milano, Italy\textsuperscript{[66]}). The purpose of the project is to compute the TTL of a mobile device and to provide such information to the user\textsuperscript{[67–69]}. For this reason, MPower is composed by two main parts:

Android App

The App purpose is to sample useful information about the device status at a certain frequency and visualize TTL and TTC to the user.

TTL computation

This part predicts the TTL based on a power model. Once the model parameters have been computed, it is possible to generate the discharge curve of the battery and so the TTL.

In Section 4.2.1 we will present MPower infrastructure, while in Section 4.2.2 the TTL power model.

4.2.1 MPower features

The idea behind MPower, a system able to predict battery life in mobile devices, was presented for the first time in 2012\textsuperscript{[67]}. This work wanted to study the behaviors of mobile devices in order to help users in energy consumption management of their devices. This idea
was manly implemented in two parts: on one hand, there is an App able to monitor the device and extract the information necessary to future analysis, on the other hand, a remote server where data are stored and the very power model is computed.

We now describe each one of MPower main operations (Figure 10):

a. When the user runs MPower for the first time, he/she is registered on the remote server with a univocal ID.

b. At this point, MPower starts collecting data necessary to the power model computation. In this phase, only a general model, computed on all the dataset users, is available. In fact, the generation of the custom power model requires some days of learning.

c. When enough data have been gathered, these are sent to the server, where they are stored and used to compute the power model. As soon as the data collected by the server reach the minimal threshold, the model is transmitted back to the App and shown to the user.

d. The server creates several power models, one of each possible configuration of available input variables. We define, as input variables, the states of the device components, such as CPU, WiFi, Bluetooth, GPS and so on. Each variable indicates whether that certain component is on or not. The set of the observed input variables represents a configuration. One model for each configuration is necessary since, in this version, MPower does not care about user’s routine, but it supposes that, once that configuration has been set, the user will not change it. As consequence, a model for each possible configuration is needed. Since all these models are available, the user may look for another model based on his/her energy demand.
Figure 10: MPower screens
e. During research phase, the user may apply some filters to take into account only the functionality he/she needs.

f. Once the models suitable for time requirements and configurations chosen by the user have been identified, the available configurations are shown. Hence, the user can select the configuration that fits his/her requirements the most.

In order to provide scalability and efficiency as the number of users increase, the server architecture was developed in a cloud environment.

### 4.2.1.1 MPower components

Here we present the main components of MPower. An high level description is shown in Figure 11. The same components may be grouped in different ways according to the reader’s interpretation. On the one hand, a vertical lecture shows the components grouped by functionality. In particular, there are four areas:

**Logging Process**

Logging Process is in charge of periodically accessing to the important information for the analysis. It relies on a series of libraries (called Sense libraries) that are meant to avoid a continuous polling from the Android API, unless the device settings have been modified.

**Communication and Security**

In order to guarantee a good level of security, a symmetric-key cryptography protocol was implemented. The device key is generated and submitted during the registration phase; it will be asked in every App-server communication.
Power Model

As previously explained, the power model (computed on the server) predicts the device TTL for each possible configuration of variables and stores such results in a JavaScript Object Notation (JSON) file, which is sent to the device. An example of such file content is reported in Table IV. The presence of multiple configurations is necessary since the App helps the user choose the best configuration according to his/her needs.

Miscellaneous

Miscellaneous area contains all the components used for events management. For instance, it allows to recognize unused components that are active or it alerts the system whenever enough data have been collected and there are the suitable conditions for broadcasting.
TABLE IV: INSTANCE OF TTL TABLE

<table>
<thead>
<tr>
<th>WiFi</th>
<th>GPS</th>
<th>Screen</th>
<th>Mobile</th>
<th>Bluetooth</th>
<th>Airplane</th>
<th>1%</th>
<th>2%</th>
<th>...</th>
<th>99%</th>
<th>100%</th>
</tr>
</thead>
<tbody>
<tr>
<td>off</td>
<td>off</td>
<td>low</td>
<td>on</td>
<td>off</td>
<td>off</td>
<td>4</td>
<td>12</td>
<td>...</td>
<td>987</td>
<td>1031</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>on</td>
<td>on</td>
<td>medium</td>
<td>off</td>
<td>off</td>
<td>off</td>
<td>1</td>
<td>4</td>
<td>...</td>
<td>598</td>
<td>632</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>on</td>
<td>off</td>
<td>medium</td>
<td>off</td>
<td>off</td>
<td>off</td>
<td>1</td>
<td>4</td>
<td>...</td>
<td>798</td>
<td>815</td>
</tr>
</tbody>
</table>

such data to the server. In fact, data transmission happens only when the mobile device is plugged in and not in use: in this way, it guarantees not to impact on the device power consumption.

On the other hand, a horizontal lecture groups components by views and classifies them in:

**Presentation layer**

This layer is in charge of handling the user interaction; more in detail, it provides TTL visualization.

**Application Logic layer**

Application Logic layer handles the internal logic related to services, communication aspects and events management.

**Hardware and Data Access layer**

Such layer contains a set of libraries that are used during device monitoring phase to access user’s settings.
4.2.2 Power Model Visualization

As we can see in Figure 10c, MPower provides two fundamental information to its users: TTL and TTC. Hence, it is clear that there are two different power models, which we will briefly present.

The TTC power model is computed using a linear regression analysis technique based only on the battery charge level. In fact, the results showed that the TTC is independent from device components (e.g. WiFi, GPS, Bluetooth and so on), but depends also on the battery charger used by the user. Different battery chargers imply different charge curves and, therefore, different TTCs. At the moment, it is not possible to infer which battery charger is being used (there are no APIs for that). For this reason, MPower collects the different charge curves and provides the user with the most likely TTC according to the charge curves collected so far.

MPower collects information about audio, battery, Bluetooth, CPU, GPS, Mobile, screen and WiFi in order to build a power model of the device. In its first release on Google PlayStore (September 2013), MPower purpose was to obtain an acceptable precision level in TTL computation given a certain device configuration. The TTL power model is computed with a regression technique on data. Power model computation has two different functionality as result. On one hand, it provides the user with TTL information, which depends on device internal states. On the other, it allows the user to easily manage his/her device battery. In fact, battery management is permitted by a very user-friendly interface that shows the TTL relative to all possible device internal states variations. The internal states considered are obviously the ones that the user can manage. For instance, the interface shows the TTL associated to...
the configurations and so the user can set the most suitable configuration according to his/her requirements. In fact, thanks to a shortcut, the user can select the chosen configuration and this will be automatically set.

4.3 Proposed Solution

The goal of this work is to propose an efficient power monitoring approach, deeply fused within Android OS, able to provide accurate and reliable information about the device status and with a negligible impact on the battery.

Our idea is to move MPower logging service from the high Java level (Application Layer and Application Framework) to an lower C/C++ OS level. In other words, outside the Android runtime environment (Figure 2). In particular, we want to implement a C++ native daemon (or service, as daemons are called in Android). We want to avoid interaction with services and Android Framework APIs. Therefore, we have to retrieve the same information MPower gets in another way.

In Linux there are two virtual filesystems: Procfs and Sysfs, which are maintained by the currently running kernel in the system[70]. Traditionally, filesystems are used to store data persistently on block devices. However, it is also possible to use filesystems to organize, present, and exchange information that is not stored on block devices, but dynamically generated by the kernel. This is the case of Procfs and Sysfs.

Procfs

The proc filesystem enables the kernel to generate information on the system state and configuration. Such information is available from normal files to users and system pro-
grams without special tools for communication with the kernel (indeed, a `cat` command is enough). These data can both be read from the kernel and be sent to it by writing to these files. This approach makes use of a virtual filesystem that generates file information “on the fly” (in other words, only when requested by read operations). Therefore, there is no need for a dedicated hard disk partition or some other block storage device with filesystems of this type.

**Sysfs**

Sysfs filesystem is a particularly important virtual filesystem that serves a similar purpose to procfs. Sysfs is usually mounted at `/sys`. It was designed to export information from the kernel into user-space at a highly structured level. However, it was not designed for direct human use (like procfs) because the information is deeply and hierarchically nested. Moreover, the files do not always contain information in American Standard Code for Information Interchange (ASCII) text form, but may use binary strings. On the other hand, such filesystem is very useful for tools that want to gather detailed information about the hardware present in a system and the topological connection between the devices.

In addition to the Procfs and Sysfs, the kernel provides many other virtual filesystems for various purposes (e.g. for the management of all devices and system resources cataloged in the form of files in hierarchically structured directories).

The logging daemon can be implemented exploiting values from files from the Procfs and Sysfs. It sounds very simple, the only complex part here is to know where the correct file for a specific
logging state is located. Moreover, another issue comes on scene; indeed, we are moving toward
device specific solution. This means that if we run the exact version of the logger on another
device, it is highly probable that some of the files in Procfs and Sysfs are not going to be
correct.

Using a native C++ daemon, instead of the MPower logger, will allow us to retrieve more
accurate information about device state and reduce battery impact. To achieve such goals, we
decided to implement our daemon as part of Android OS itself; therefore, we want to create
a custom version of Android able to run our daemon as part of the OS. Finally, we want to
integrate our daemon with MPower, so that MPower can rely on data collected by our daemon
and avoid to collect them directly. In Chapter 5, we will describe how we manage to create our
C++ daemon, compile it within Android source code and integrate it with MPower.
CHAPTER 5

IMPLEMENTATION

The solution we propose is to move MPower logging service from Java Application level to a lower level, so that it can both run outside Dalvik and avoid interaction with Android services; this goal can be achieve by exploiting virtual filesystems (proc and sys) and reading the device state from their files. Therefore, our purpose is to create a native C++ daemon able to run as part of Android OS. To do so, we both need to customize Android source code and choose a target device. The device we target is Samsung Galaxy SIII (international version). First of all, we need to understand why moving the MPower logging service to a lower level may make sense and what overhead our solution may avoid. For this reason, we want to briefly contextualize the problem in Android architecture (Section 5.1). Then, in Section 5.2, we define the rooting process necessary to install a custom version of Android on a device, then, in Section 5.3 we present the logger implementation. Finally, in Section 5.4, we briefly describe how we integrated our daemon with MPower App.

5.1 Context background

A contextualization of how Android manages power consumption and which part of Android architecture MPower interacts with is required before moving to the implementation itself of our daemon. In Section 5.1.1, we analyze Android Power Management Section 5.1.1, while
in Section 5.1.2 the Android architecture is explained (in particular, the parts related to our work).

5.1.1 Power management in Android

Figure 12: Android Power Management Application Framework

In Section 3.1, we have seen how power consumption is managed in computers. In particular, we explained that OSes like Linux implements functionality able to place the entire system in a
low power state, and, at the same time, still able to guarantee a low response latency. However, the power consumed in this low-power sleep mode becomes critical when we consider battery-powered devices like smartphones. In such configuration, the device is not meant to be actively used by the user and it will stay in this state for the majority of the time, being however still connected and able to receive calls and so on. For this reason, Android implements a lightweight version of ACPI driver optimized for embedded systems[72] and provides the Android Power Management Application Framework. This Java framework behaves like a driver on the top of SPM (Figure 12), makes a JNI call to native kernel libraries and supplies an aggressive power management policy. In details, the assumption is: the device must not consume power if no power is required by any software component. However, should user level Apps or services require CPU resources, the OS cannot put the hardware into sleep or suspend state. Such mechanism is called wake lock[73]. In order to access power management system and require a wake lock, user space Apps use the PowerManager class. In this way, the Apps notify the OS they need a certain functionality and avoid the OS putting the device in idle state.

In Figure 13, the three most important system states are shown: the AWAKE state is reached after a touch user activity event or after an App acquired a full wake lock. Then, a power key pressing event, as well as a timeout, makes the system go in the NOTIFICATION state. Finally, the device enters SLEEP state when all partial locks are released. The more the Apps are well designed (i.e. they acquire just the necessary resources), the more this power saving system is efficient. If the App holds a PARTIAL_WAKE_LOCK, the CPU will continue to run, regardless of any display timeouts, screen state and power key pressing event. In all other wake locks
CHAPTER 3. MOBILE OPERATING SYSTEMS BACKGROUND

Figure 3.8: System cycle for flow of events during process life. [8]

User space applications use the "PowerManager" class to access the Power Manager system service and require a wake lock: in this way, they declare to the operating system that they need a particular functionality and force it not to put the device in an idle state.

Figure 3.8 shows the three most significantly system states: in case of a touch screen or keyboard event, as well as if an application acquires a full wake lock, Android enters or keeps in the **AWAKE** state. A timeout or a power key pressing event can then force the system in the **NOTIFICATION** state. If all partial locks are then released, the device will finally go into the **SLEEP** state.

This state machine represents a more efficient power saving behavior for a mobile device as soon as applications are well designed in order to acquire just the resources (i.e. **SCREEN_DIM_WAKE_LOCK**, **SCREEN_BRIGHT_WAKE_LOCK**, and **FULL_WAKE_LOCK**), the CPU will still run, but the user can put the device to sleep using the power button. Usually, partial wake locks are used by background services; in this way, the device is forced to stay awake until that particular task has been completed.

5.1.2 **Android Architecture**

In this section, we provide a bottom-up overview of Android OS architecture[4] (Figure 14). Even if it would be interesting to analyze every part of Android architecture, we will focus only
on those parts essential for our work. In particular, we will examine Android Kernel, Android Java level and System Services.

5.1.2.1 **Android Kernel**

Although it is based on Linux “vanilla” kernel, Android OS is quite different from a standard Linux system, from both a user and kernel space point of view\textsuperscript{[74]}. For instance, the wake
lock mechanism we explained in Section 5.1.1 is a prerogative of Android Kernel. Another fascinating part of Android is the low-memory killer in addition to the default kernel Out of Memory (OOM) killer. However, here we are interested in a particular mechanism called **Binder**.

### 5.1.2.2 Binder

Binder is the Android lightweight RPC/IPC mechanism. Android communication across components is quite different from the one implemented in Linux, which typically uses socket or System V IPC\(^\text{[70]}\). Moreover, Linux IPC mechanism is not suitable for Android, since it is not so efficient. On the other hand, Android uses the in-kernel Binder mechanism. Essentially, Binder attempts to provide remote object invocation capabilities on top of a classic OS, in order to permit the developers to deal with remote services as objects. Binder is a cornerstone of Android architecture (Figure 14). It allows Apps to interact with Android System Services. However, App developers do not use the Binder mechanism directly. In fact, they must define and interact with interfaces using Android Interface Definition Language (AIDL). Such interface definitions are then processed by the AIDL tool, which generates the proper stubs and marshaling/unmarshaling code required to transfer objects and data back and forth using the Binder mechanism. The in-kernel driver part of the Binder mechanism is a character driver used to broadcast parcels of data between the communicating parties using calls to \texttt{ioctl()}, a well-known system call for device-specific input/output operations and other operations which cannot be expressed by regular system calls. Just like Linux, Android has its own shared memory mechanism. In particular, Android uses a file-based shared memory system called Android
Shared Memory (ASHMEM). Usually, a first process creates a shared memory region using ASHMEM, and uses Binder to share the corresponding file descriptor with other processes it wishes to share the region with. Many System Services (e.g. Surface Flinger, Audio Flinger, etc.) communicate through ASHMEM rather than directly.

5.1.2.3 Dalvik

The Dalvik VM\cite{75}, designed by Dan Bornstein, is one of the main components of Android architecture. The Dalvik is the Android clean-room byte-code interpreter implementation used as a replacement for the Sun Java VM. Differently from Sun Java, Dalvik does not interpret .class files, but .dex files, which are generated from .class files by the dx utility. Java bytecode is converted into Dalvik bytecode, which is designed for systems constrained in terms of memory and processor speed. These .dex files may be modified again when installed onto a mobile device, in favor of a further optimization.

Dalvik acts like a middle tier; indeed it is the access point to all the system services and the underlying hardware. Moreover, developers have an abstraction layer that ensure they do not need to worry about a particular hardware implementation since the OS isolates Apps from each other and from the system. Every process hosts a different Dalvik instance that sandbox a single application. Hence, no App is able to execute actions that could interfere with other Apps, the OS or the user. Besides, every critical operation (e.g. connect to the network and so on) requires an explicit permission. On the other hand, all the memory and process lifetime management are left to the Android Runtime. It is responsible for stopping and killing processes to free resources for high-priority applications.
5.1.2.4 System Services

The Android System Services are built on Binder mechanism. The most important Service is the System Server, whose components all run under the same process, and which is mostly made up of Java-coded services with two services written in C/C++ (Figure 15). The System Server also includes some native code access through JNI. In this way, some Java-based services may interface to Android lower layers. Media Service contains other services that are C/C++-based and packaged alongside media-related components. Finally, the Phone service is separated from the rest. All these processes that house the whole Android System Services, appear to operate independently to anyone connecting to their services through Binder.

Figure 15: Android System Services
All the accesses to the services are done through the Service Manager. Indeed, if a service is not registered with the System Manager, it is invisible to the system. When the System Server starts, it registers every single service it instantiates with the Service Manager. When an App wants to communicate with a service, it first asks the Service Manager for a handle to that certain service and then invokes that service methods. On the other hand, a call to a service component running within an app goes directly through Binder and is not looked up through the Service Manager. We are interested in System Services since it is the most direct way to collect information about the device state (also MPower uses them to collect the device components status). However, we have just seen that, when an App invokes a service, the computation overhead is not negligible (Figure 16). This may cause an higher impact on battery power consumption and slow down and complicate the logging process. That is why we want to move MPower Logging Service to a lower level, in order to avoid such complexity. The first step is this direction is to root our device, so that we can insert our daemon as part Android OS.

5.2 Rooting

“Rooting” of a phone means obtaining a root access. It is called rooting, because the term “ROOT” in the UNIX world is related with the file system and the permissions. The branches of the file systems and users resemble an inverted tree: the root of the file system is the beginning of all the files and directories, the root of the permissions is the beginning of all the permissions. Therefore once we become root user on the phone, we can modify any file, and we can flash another version of Android. The phones that are shipped on the market have
a protection based on Original Equipment Manufacturer (OEM). This means that, when one buys a phone from, for instance, Samsung, the entire device is branded; even the software of the device. The protection is done from the carriers or from the manufactures, since they want to protect the revenue and maintain control of the device. The protection usually consists in effectively locking the start-up process of the Android OS.

The process of booting the device in the mobile world is called bootstrapping, and describes what the device does when it is powered ON. After the power ON, a small piece of code located on a memory chip initializes the memory and the CPU. It checks some hardware features, and loads the first part of the OS into the memory. This code is referred as a bootloader. When the bootloader is encrypted, the device is locked.
Bootloading is typically a two-part process: it makes use of a primary and a secondary bootloader. The primary bootloader is hardcoded on an Application-Specific Integrated Circuit (ASIC) inside the device. These hardcoded instructions load the secondary bootloader into memory and tell it where the memory, CPU and OS are located and how they can be accessed.

The goal of this section is to add a custom bootloader. The custom bootloader is a secondary bootloader, that allows to gain access to the file system with more control than with an OEM bootloader. With a custom bootloader, we can replace the original OS files with customizations, like a new user interface or a new kernel: this is the reason why we need root access.

Let us explain the steps of the bootloading process:

1. Special code in the boot ROM locates the first-stage bootloader and loads it into memory.

2. The first-stage bootloader loads the second-stage bootloader after initializing some memory and getting the hardware ready. In this step the bootloader checks the security flag. If the flag is S-ON, then it will load only signed (official) kernels. If the flag is S-OFF the bootloader does not check for signatures. By setting S-OFF other security locks are pulled down, like making the file system writable, or enabling the installation of custom recovery.

3. The bootloader loads a Linux kernel and customizations into memory.

4. The last step is the Initialization (INIT) process. The INIT process is the father of all other processes that run on the device. It initializes all the processes necessary for basic hardware access and device functionality.
The step number 2 is the most important, because we would like to load our custom boot-
loader (in that step), which will allow a custom kernel to be loaded. Therefore, in the way of rooting we would like to set the security flag to off (S-OFF). On some devices this is not possible, so workarounds need to be considered for achieving such a customization.

In our case, we are interesting of just obtaining root access. Because the purpose is to create a C native daemon application that will log the state of the phone, the rooting of the device is just the first step.

In the following subsection, the rooting process will be explained.

5.2.1 How to root

As we want to root a particular device, the standard procedure is to find a tool for rooting. Otherwise it would require a lot of knowledge of the Android OS and of the particular device: a manual procedure may brick the device if we do something wrong. In the world of rooting the term “brick” is very used, and it means to stuck the device. After the device is bricked, only the official manufacture repairers can fix it.

Usually, when people want to root a phone, it is because they want to flash another version of Android, which may not work. Therefore after the rooting procedure, it is very recommended to install a custom recovery application. Every Android phone comes with a recovery routine. The recovery is a separate, standalone piece of code on a partition that can be booted in order to update Android and maintain the device. The problem is that the original recovery makes updates only to the official update packages that are signed with the OEM digital signature.
We would like to install another recovery, usually referred as custom recovery, that allows to omit this restriction. Also the custom recovery provides a set of very useful features:

- resetting a device to factory settings,
- clearing the data cache,
- making complete backup and restore,
- installing an update of the Android OS.

The custom recovery can be installed only after the phone is rooted. The rooting process together with installing a custom recovery is considered as a standard procedure. Before flashing the new custom OS, we can do a complete backup, which is stored on the Secure Digital (SD) card. Then after installing the new OS version, if something does not work, we can simply restore the previous version.

On the Internet there is a big community of people who are very knowledgeable in Android. One of this is the XDA-Developers web portal\cite{xda-developers}: on this portal there are information for almost every present device. People are testing tools, reporting bugs, making suggestions, etc. We relied on it to find a suitable tool for our purpose.

For the Samsung Galaxy S III, the Heimdall Suite was used\cite{heimdall-suite}. Heimdall is a cross-platform open-source tool suite used to flash ROMs onto Samsung mobile devices. In order to use this tool, we have to run the device under the Download Mode. This mode is a unique boot mode of Samsung devices which is very similar to Fastboot Mode. Fastboot Mode is a protocol used to directly update the flash filesystem in Android devices from a host over Universal Serial Bus.
It allows flashing of unsigned partition images. It is disabled on almost all production devices since USB support is disabled in the bootloader. Thanks to this tool, we are able to install the custom recovery application. In particular, we used the popular ClockworkMod recovery, version 6.0.4.3. (Figure 17).

5.2.2 Custom ROM: the CyanogenMod

All Android files (i.e kernel, native Apps, etc.) are packed into a single package. Very often this packages are called ROM. Technically, the memory where we install the OS is a Non-Volatile Storage Technology (NAND), which is not a ROM. However, this term has become a
standard name. The installation process is called *flashing*. There are many Android Custom ROM, for instance Paranoid Android\[79\], Shiny ROM\[80\] and so on.

CyanogenMod\[81\] is a customized, aftermarket ROM distribution for several Android devices (smartphones and tablets). Based on the AOSP, CyanogenMod is designed to increase performance and reliability over Android-based ROMs released by vendors and carriers. Moreover, CyanogenMod offers the most barebone Android experience coupled with a variety of features and enhancements that are not currently found in standard versions of Android. There are mainly four releases of CyanogenMod ROMs:

**Stable release** is the stable and tested version of the ROM proven to be mostly bug free and suitable for daily use.

**Release Candidate** build may not be the final version, but a variant that has no fatal flaws or bugs, on the stabilization stages to become the final product that is the Stable version.

**M-series releases** behave similar to the Release Candidate, but is considered “stable” for the users.

**Nightly release** is an automatic build from a codebase, without any developer-specific changes.

One may ask why we need a Custom ROM like CyanogenMod when Android is open source. The answer is that, in order to install Android OS on a device, we also need all the drivers and have to know how to set the environment for that particular device. Of course, Android source code is available and easily downloadable (as referred in Chapter 2), but, to flash it on a device (Samsung Galaxy SIII in our case), we also need all the drivers for the device. The drivers are
produced by the manufacturers, and they are not available publicly. That is the reason we need a Custom ROM.

CyanogenMod uses the Android original source code, and many developers are contributing in the project by porting the OS for specific devices. This means that the Android versions available on CyanogenMod website contain all the drivers needed for that particular device. We chose to use the CyanogenMod Custom ROM because it supports Samsung Galaxy SIII and it is very well documented (both for installing and building it for SIII). Our idea is to customize the CyanogenMod by adding a C++ native daemon that will run inside Android OS level and outside of the Dalvik/Java level. The only way to accomplish this is by downloading the source code of the CyanogenMod for the Samsung Galaxy SIII, adding the daemon, rebuilding the OS and flashing it on the device. In this work, we used CyanogenMod version 11, based on Android K.

The CyanogenMod can be built only under Linux. In particular, it is necessary Ubuntu\textsuperscript{82} 12.04 Long Term Support (LTS), 64-bit version. Moreover, particular versions of the tools needed are required (e.g. Java version 6). Two of the main tools we used are Vagrant\textsuperscript{83} and Virtual Box\textsuperscript{84}. Virtual Box is a well-known virtualization tool, while Vagrant is a tool for building complete development environments. In particular, we used Vagrant in order create and configure a VM that contained all the tools we needed to compile Android.

As we said previously, CyanogenMod is based on Android official source code. Therefore, CyanogenMod follows the same directory structure as the AOSP. In Table V, all the main directories in AOSP and their short meaning are presented\textsuperscript{[4]}. 
In our case, the most important directories are the ones of the device and the system. In the device directory we have the device based files, while the system directory contains the system files: here is where we would like to add our C++ native daemon. The Android build system is organized using Android makefiles (Android.mk). The difference from Linux is that here the build runtime does not search recursively for all the makefiles, instead it collects all the makefiles, assembles one large makefile, and then starts the build. In Figure 18 is presented the general diagram of the build system. Everything starts from the envsetup.sh. When we run this script we can see all the possible configurations for which the build can be done (i.e. for
the emulator, our specific device, etc.). Once we make the decision, the environment variables are initialized, and then used during the build. Of course Android compilation process is very fascinating, but we will not focus on how to compile Android source code. Instead, we are interested in how we can add our native daemon inside Android source code.

In Android, there are several module build templates. It is important to stress that Android build modules have nothing to do with kernel modules. Regarding of Android build system, a module is any component of the AOSP that needs to be built. Therefore, it might be a binary, an App package and so on. Build templates are provided so that module authors can get their

---

Figure 18: Android build system[^4]
TABLE VI: SOME OF MODULE BUILD TEMPLATES

<table>
<thead>
<tr>
<th>Variable</th>
<th>Template</th>
<th>What It Builds</th>
<th>Most Notable Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUILD_EXECUTABLE</td>
<td>executable.mk</td>
<td>Target binaries</td>
<td>Native commands and daemons</td>
</tr>
<tr>
<td>BUILD_HOST_EXECUTABLE</td>
<td>host_executable.mk</td>
<td>Host binaries</td>
<td>Development tools</td>
</tr>
<tr>
<td>BUILD_RAW_EXECUTABLE</td>
<td>raw_executable.mk</td>
<td>Target binaries that run on bare metal</td>
<td>Code in bootloader/ directory</td>
</tr>
<tr>
<td>BUILD_JAVA_LIBRARY</td>
<td>java_library.mk</td>
<td>Target Java libraries</td>
<td>Apache Harmony and Android Framework</td>
</tr>
<tr>
<td>BUILD_STATIC_JAVA_LIBRARY</td>
<td>static_java_library.mk</td>
<td>Target static Java libraries</td>
<td>N/A, few modules use this</td>
</tr>
<tr>
<td>BUILD_HOST_JAVA_LIBRARY</td>
<td>host_java_library.mk</td>
<td>Host Java libraries</td>
<td>Development tools</td>
</tr>
<tr>
<td>BUILD_SHARED_LIBRARY</td>
<td>shared_library.mk</td>
<td>Target shared libraries</td>
<td>A vast number of modules, including many in external/ and frameworks/base/</td>
</tr>
<tr>
<td>BUILD_STATIC_LIBRARY</td>
<td>static_library.mk</td>
<td>Target static libraries</td>
<td>A vast number of modules, including many in external/</td>
</tr>
</tbody>
</table>

modules built appropriately. Each template is designed for a specific type of module. Some of these templates are presented in Table VI.

Since we want to insert a native daemon logger in Android source code, the module we are looking for is BUILD_EXECUTABLE module. Indeed, this module template builds binaries for native commands and daemons. Usually this modules are placed in a specific folder of the AOSP code (/system/core directory), and, after the build, they are placed in the /bin directory.

The build system of Android is hierarchical and is organized with makefiles. Each module must have a makefile (Android.mk), so that the build process knows how to compile the module and where to place the output. The module build templates specify the structure of the module makefiles. In our case, the makefile has this structure:
Listing 5.1: Module makefile structure

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>LOCAL_PATH := $(call my-dir)</code></td>
</tr>
<tr>
<td>2</td>
<td><code>include $(CLEAR_VARS)</code></td>
</tr>
<tr>
<td>3</td>
<td><code>LOCAL_MODULE := # module name</code></td>
</tr>
<tr>
<td>4</td>
<td><code>LOCAL_MODULE_TAGS := optional</code></td>
</tr>
<tr>
<td>5</td>
<td><code>LOCAL_SRC_FILES := # source code</code></td>
</tr>
<tr>
<td>6</td>
<td><code>LOCAL_SHARED_LIBRARIES := # libraries used</code></td>
</tr>
<tr>
<td>7</td>
<td><code>include $(BUILD_EXECUTABLE)</code></td>
</tr>
</tbody>
</table>

`LOCAL_PATH` specifies where the module is located. When the runtime process comes to this file, it calls the `my-dir` function from the definitions of makefile. The function returns the local path for the module, and path variable is initialized. This makefile will be placed as part of a bigger makefile, so when the build starts, it will know where to find our module.

`include $(CLEAR_VARS)` clears all previously set `LOCAL_*` variables that might have been set for other modules.

`LOCAL_MODULE` is the global name of the module.

`LOCAL_MODULE_TAGS` defines tags for build, in order to determine whether that module has to be installed in that source code build.

`LOCAL_SRC_FILES` specifies all the source files that need to be included.

`LOCAL_SHARED_LIBRARIES` specifies some local libraries that are used in this module.

`include $(BUILD_EXECUTABLE)` invokes the build template that corresponds to the current module type.

These build templates allow `Android.mk` files to be usually fairly lightweight. Now we have to include our module in the module list of our device. Indeed, if we build now the Android
source code, the module will be build but it will not be included in the module list of the Samsung Galaxy SIII. Hence, we have to add our module in the module list of the device, and this can be done by modifying the product makefile of the device. We opened the makefile i9300.mk and added our daemon package to the list of PRODUCT_PACKAGES, which allows us to specify packages we would like to have included in this product, in addiction to those specified in the products we are already inheriting from. In this way, our package will be included in the final image generated.

Our module is a native daemon, therefore we want the daemon to be automatically started when the device boots. In Android, as well as in Unix-based OS, the INIT process starts all the processes and services. The INIT process is configured through the init.rc script file. In this file it is specified what needs to be done when the initialization takes place. We have to edit the global init.rc script to set the module to be start on boot time. At the bottom of the script are added all the native daemons, therefore we just need to add this code at the very end of the script:

```
1 ...  
2 service our_module /system/bin/our_module
3   oneshot
```

The code is very short and simple, it just says: start our daemon that is located in the /system/bin/ directory (build executables after the build go in the /bin folder). The oneshot command specifies: do not restart the process if the process exits somehow. We will see in Section 5.3 that, in the standard daemon Linux template, a daemon process always forks in
TABLE VII: DATA COLLECTED BY THE DAEMON

<table>
<thead>
<tr>
<th>Audio</th>
<th>Battery</th>
<th>Bluetooth</th>
<th>CPU(s)</th>
<th>GPS</th>
<th>Mobile</th>
<th>Screen</th>
<th>WiFi</th>
</tr>
</thead>
<tbody>
<tr>
<td>music_active</td>
<td>on_charge</td>
<td>is_on</td>
<td>max_freq</td>
<td>is_on</td>
<td>is_on</td>
<td>is_on</td>
<td>is_on</td>
</tr>
<tr>
<td>speaker_on</td>
<td>temperature</td>
<td>state</td>
<td>min_freq</td>
<td>state</td>
<td>activity</td>
<td>brightness_mode</td>
<td>is_connected</td>
</tr>
<tr>
<td>music_volume</td>
<td>voltage</td>
<td></td>
<td>current_freq</td>
<td></td>
<td>net_type</td>
<td>brightness_value</td>
<td></td>
</tr>
<tr>
<td>ring_volume</td>
<td>percentage</td>
<td></td>
<td>max_scaling_freq</td>
<td></td>
<td>signal_strength</td>
<td>width</td>
<td></td>
</tr>
<tr>
<td></td>
<td>technology</td>
<td></td>
<td>min_scaling_freq</td>
<td></td>
<td>tx_bytes</td>
<td>height</td>
<td>link_speed</td>
</tr>
<tr>
<td></td>
<td>health</td>
<td></td>
<td>governor</td>
<td></td>
<td>rx_bytes</td>
<td>refresh_rate</td>
<td>tx_bytes</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>usage</td>
<td></td>
<td>call_state</td>
<td>orientation</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>cpu_id</td>
<td></td>
<td>airplane_state</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

another child process. Actually, the main code runs in the child process. The parent process simply terminates after the fork. If we do not specify oneshot, when the parent process terminates, Android system will restart the process again. The process will terminate again, and therefore it will be restarted again from the system. This will continue in an endless cycle.

Now, everything is configured for our module. We can build Android source code and flash it on the device. In Section 5.3, we will finally focus on our daemon logger implementation.

5.3 Daemon logger implementation

The final goal of this work is transferring the high level Java logger of MPower App into the C++ native daemon. The first goal of the project was to understand how to root, build and modify the CyanogenMod custom ROM. As we said in Section 4.3, our daemon is meant to read the state of the device from Procfs and Sysfs.

For the logger daemon we used the standard Linux daemon template. In this template, we simply fork another process, where the logger daemon will run, and the parent process terminates. For this reason we used the command oneshot inside the init.rc script. Besides, for
the daemon process it is very common to disable the standard input and output, and to enable permissions of the folder in which the daemon is running. The daemon is a service application that usually never ends. Therefore inside the daemon, there is one everlasting loop cycle. Each 10 seconds (just like MPower) the daemon reads and saves the state of the device in a log file. Every day, the daemon produces a different log file. The data are stored on the log file in JSON format. The data collected by the daemon are listed in Table VII. Moreover, the daemon collects information about the manufacturer ID, the product ID and the device ID.

One important and interesting part of the daemon logger implementation is the configuration file. In this file, we list the paths to all the files we have to read in order to collect all the information we need. Here a snippet of such file:

```
Listing 5.3: Snippet of configuration file

...  
//----- QUERY PATHS FOR THE CPU ------
//----- CPUINFO, ------//
char Q_CPU_MAXF[] = "/sys/devices/system/cpu/cpu0/cpufreq/cpuinfo_max_freq";
char Q_CPU_MINF[] = "/sys/devices/system/cpu/cpu0/cpufreq/cpuinfo_min_freq";
char Q_CPU_CURRF[] = "/sys/devices/system/cpu/cpu0/cpufreq/scaling_cur_freq";
char Q_CPU_MAX_SF[] = "/sys/devices/system/cpu/cpu0/cpufreq/scaling_max_freq";
char Q_CPU_MIN_SF[] = "/sys/devices/system/cpu/cpu0/cpufreq/scaling_min_freq";
char Q_CPU_GOVERNOR[] = "/sys/devices/system/cpu/cpu0/cpufreq/scaling_governor";
char Q_CPU_USAGE[] = "/sys/devices/system/cpu/cpu0/cpufreq/cpuinfo_cur_freq";
char Q_CPU_ID[] = "/sys/devices/system/cpu/cpu0/cpufreq/affected_cpus";
//----- END CPU ------
...  
```
This file is particularly important because it guarantees the portability of our daemon. Indeed, if we want to use the daemon with another device, it is just sufficient to change the paths to device state files (sometimes it is not necessary since some of the them are common for many devices).

During daemon execution, we are carefully logging all its the states. That is very important when writing such types of application. If something is wrong with the daemon or we are not logging all the states, it is impossible to figure out what is the bug of our daemon. In Android, there is an integrated logging system. By calling the `SLOGI` function, we are directly logging in the main log of the Android. Each module have to define its own logging tag. Once we have defined our logging tag, we can use the Android Debug Bridge (ADB) library to examine the log of the device. In this way, we can verify our daemon state and check whether there is any bug or not. Finally, it is crucial to prevent Android OS suspend the daemon execution when the system is put to sleep. For this reason, our daemon acquires a wake lock before logging data and releases it after they have been written onto the log file.

### 5.4 Integration with MPower

Finally, we describe how we integrated our daemon logger with MPower App. At the moment, we just inserted our daemon as alternative to MPower logging service. Further possible integration will be presented in Section 7.3.

When MPower runs, it checks if our daemon is already running and, if so, it does not start its own logging service. To get such information about our daemon running state, we thought about using JNI to provide an elegant and efficient integration. We have already explained
that JNI allows for coexistence of Java and native C/C++ code. We used the NDK in order to create a C library able to check whether our daemon is running or not. Then, JNI creates a wrapper of the C library for Java level; in this way, the C function we implemented in our library can be called as a normal Java method by just importing the package.
CHAPTER 6

RESULTS

Here, we want to present the experimental results of our work. In this work, we focus on the comparison between the battery impact of MPower and our daemon. Moreover, we tested whether our daemon can log at a frequency higher than the one we found as MPower upper bound or not. We used *MatLab*\[^{[85]}\] as statistical analysis environment. First of all, we want to present the results of MPower impact on battery measured at OS level. Then, we introduce our testing infrastructure. Indeed, in order to validate the results (in particular, for the battery impact), we used a power monitor able to measure the current drained by the device in three testing configuration (Section 6.2). Once we gathered all the necessary data, we compared such data and extracted the results of this work (Section 6.3). Finally, we tested whether our daemon can log faster than MPower or not (Section 6.4).

6.1 MPower OS level tests

Here, we show the results collected by MPower developers, along with us, about MPower impact on battery life of the monitored device. In order to estimate if and how much the such App influences the power consumption of the device with respect to its normal behavior, we compared the discharge curves of the monitored devices with and without MPower installed on them. This is not a trivial task, since the device battery life is significantly influenced by both external environment and users usage patterns. We ran tests on a small number of devices
(chosen to be a testing platforms the most representative possible) and defined significant case studies, in a controlled and stable environment. Finally, we performed a hypothesis test in order to prove, with a certain level of confidence (1%), that the MPower does not influence the discharging behavior of the device. Such hypothesis tests assumed that both populations have a normal distribution and same variance. Populations are composed by the times elapsed between the fully charged battery level and the lowest battery level reached in both the scenarios (i.e. without and with MPower). The null hypothesis is: the elapsed time is equal with and without the MPower installed. We chose to rely on the OS information on the current battery level to monitor battery discharge curves. Its accuracy was enough for such measurements, since we just need to show that the logging operations performed by MPower do not significantly impact on the overall system power consumption. For this reason, we designed a low battery impact custom monitoring tool able to trace, with enough precision the different battery discharge curves experienced during the tests. Besides, we were not interested in observing the whole discharging curve, since it may be considered linear for lithium-ion batteries. Non linearity are expected the more the battery levels reaches a low level (i.e. under 20% approximately). That is why all the tests were performed from a full charged battery. The test length were defined according to the scenario, in order to observe the discharging phenomenon with enough accuracy.

6.1.1 Target devices

The tests were performed on real devices, since the Android SDK emulator was not suitable for such purposes. In particular, we chose two devices in order to cope with different hardware
platforms and different Android OS versions. The first device chosen was a *LG Optimus One P500* smartphone\[^{[86]}\], running Android 2.3 Gingerbread, while the second was a *Galaxy Tab 2 7.0* tablet\[^{[87]}\], running Android 4.1.1 Jelly Bean (Table VIII shows a detailed comparison between these platforms). Such devices were chosen since their hardware characteristics and their OS version were similar to those used by entry-level (the former) and mid-range (the latter) smartphones and tablets available, in that period, on the market.

All tests were performed on both devices, analyzing their battery discharge curve in the same testing conditions, with and without the MPower running in the background. The idea was to detect the difference between the slopes of these curves, in order to estimate the impact of MPower on power consumption.

### 6.1.2 Case studies

We have seen so far that the most significant aspects that affect the device battery life are:

<table>
<thead>
<tr>
<th></th>
<th>LG P500</th>
<th>Galaxy Tab</th>
</tr>
</thead>
<tbody>
<tr>
<td>Android version</td>
<td>2.3 Gingerbread</td>
<td>4.1.1 Jelly Bean</td>
</tr>
<tr>
<td>Processor</td>
<td>600 MHz, single-core</td>
<td>1.0 GHz, dual-core</td>
</tr>
<tr>
<td>Memory</td>
<td>418 MB RAM</td>
<td>1 GB RAM</td>
</tr>
<tr>
<td>Display technology</td>
<td>TFT LCD capacitive</td>
<td>PLS TFT LCD capacitive</td>
</tr>
<tr>
<td>Display size</td>
<td>320x480 px, 3.2 inches</td>
<td>1024x600 px, 7.0 inches</td>
</tr>
<tr>
<td>Battery</td>
<td>1.500 mAh</td>
<td>4.000 mAh Li-Ion</td>
</tr>
<tr>
<td>Wi-Fi</td>
<td>802.11 b/g</td>
<td>802.11b/g/n</td>
</tr>
<tr>
<td>Bluetooth</td>
<td>2.1</td>
<td>3.0</td>
</tr>
<tr>
<td>GPS</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Mobile network (faster)</td>
<td>HSDPA</td>
<td>HSPA+</td>
</tr>
</tbody>
</table>
- its internal state (e.g. the WiFi module state, the screen brightness, etc.);
- the environment surrounding it (e.g. the strength of the wireless network signal, the temperature, etc.);
- the users usage patterns (e.g. a user can use it just to make phone calls, another one can mainly use it like a media player, etc.).

For this reason, we decided to run their tests in three different scenarios:

**Multimedia case study**

The multimedia case study simulates the case in which the user is using her smartphone as a multimedia device to play a video in streaming. Hence, the screen is always on, with a high brightness level. A high speed network connection is needed, so we used the Wi-Fi network interface to guarantee a good quality level. The duration for this case study was stated to be one hour of playing, since it was enough to observe the discharging phenomenon with enough accuracy.

**Calling case study**

The calling case study simulates the case in which the user is mainly using her smartphone to make phone calls. No WiFi or mobile data connections are available and the display is turned off. A standard one-hour GSM call is performed, reasonably long enough to observe the discharging phenomenon with sufficient accuracy.

**Idle case study**

The idle case study represents the case in which the user is not using her smartphone at
all. Therefore, display is always off, like WiFi and Bluetooth network interfaces. Only mobile data connection has been left enabled. This case study needs a longer testing period, since in such a condition the device consumes the less possible. 18 hours seemed enough to observe the discharging phenomenon with enough accuracy. Besides, the we decided to enable the email synchronization with the remote Google service provider, so that it was possible to simulate a lazy email activity.

Table IX shows a comparison between the case studies. The first two are the most representatives, since we simulate real usage patterns. The last one is the worst case, where, approximately, only MPower contributes to power consumption. All tests were run on both devices for every case study. Measurements were repeated with and without MPower installed on the devices, finally comparing the battery discharge curves.

### 6.1.3 Results of MPower tests

Here were reports MPower experimental results. In the Figures, Thinner lines denotes a region including 95% of the measurements (twice the standard deviation), while the ticker ones

<table>
<thead>
<tr>
<th>Multimedia</th>
<th>Calling</th>
<th>Idle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Display bright.</td>
<td>High</td>
<td>Off</td>
</tr>
<tr>
<td>Audio level</td>
<td>Medium</td>
<td>Medium</td>
</tr>
<tr>
<td>WiFi net.</td>
<td>Connected</td>
<td>Off</td>
</tr>
<tr>
<td>Mobile net.</td>
<td>Off</td>
<td>Off</td>
</tr>
<tr>
<td>Mail sync.</td>
<td>Enabled</td>
<td>Disabled</td>
</tr>
<tr>
<td>Duration</td>
<td>1 hour</td>
<td>1 hour</td>
</tr>
<tr>
<td></td>
<td></td>
<td>18 hours</td>
</tr>
</tbody>
</table>
Figure 19: Average discharging curves in the multimedia case study

represent the average of the discharging curves obtained performing the same test multiple times. The blue curves represent the case of MPower not installed on the device, while the red curves represent the case with MPower installed.

Multimedia case study

Figure 19a shows the results of the tests performed on the LG P500. As we can see at a glance, there are no significant differences in such a "stressed" scenario.

The same happened with the Samsung Galaxy Tab (Figure 19b. The average discharging curves are almost completely overlapped and the tests with the MPower App installed even seems to better perform, probably due to an little error in the measurements or to a non observable phenomenon.
Calling case study

This case study was the one most influenced by the external conditions, since the GSM signal strength changes as soon as the device is moving or the weather changes. Moreover, the power consumed depends on the conversation hold by the phone (e.g. a noisy environment implies more data to be sent). Therefore, the tests were performed in a completely silent environment and without moving the devices. Figure 20a shows the results of the tests performed on the LG P500. In the beginning, the two average behaviors are completely overlapped, then the tests with the MPower installed even seems to better perform: again, this is probably due to an error in the measurements or to a non observable phenomenon.
A completely analogous scenario is experienced with the Samsung Galaxy Tab (Figure 20b). Even in this case study, the presence of the MPower application seems not to worsen the battery life of the device.

Idle case study

This is probably the worst case scenario for the tests, since the device was left in an almost completely idle state for about 18 hours. During such a long time period, MPower performed about 6480 logging operations and the only other user application running on the devices is the mail client App.

Figure 21a shows the results of the tests performed on the LG P500. The MPower App seems to reasonably affect the device power consumption, since it is the only significant
activity performed by the device during the testing period.

It seems not to be the case of the Samsung Galaxy Tab (Figure 21b). Indeed, in this case, the MPower application seems not to affect the device power consumption.

The obtained results for the hypothesis tests are summarized in Table X. There is a strong statistical evidence (with confidence $\alpha = 1\%$) that MPower influenced the discharging behavior only for the idle case study and only for the LG P500 testing platform. Such test cases were useful to understand which is the impact of the single MPower App on the overall system in a controlled environment. Therefore, we showed that the presence of the MPower App does not impact on the device battery life in the most significant scenarios. The MPower application seems to consume about the 4.08% more only in the idle scenario, on the LG P500 platform.

This was the work we have done on MPower at OS level. Now we want to prove if such results are consistent using a testing infrastructure able to measure the current drained by the device. In this way, we will provide more precise and accurate information about the impact on battery of life of both MPower and our daemon.

<table>
<thead>
<tr>
<th></th>
<th>Multimedia</th>
<th>Calling</th>
<th>Idle</th>
</tr>
</thead>
<tbody>
<tr>
<td>LG P500</td>
<td>$H_0$</td>
<td>$H_0$</td>
<td>$H_1$</td>
</tr>
<tr>
<td>Samsung Galaxy Tab</td>
<td>$H_0$</td>
<td>$H_0$</td>
<td>$H_0$</td>
</tr>
</tbody>
</table>
6.2 Monsoon Power Monitor

The Power Monitor hardware\cite{footnote11} and Power Tool software provide a robust power measurement solution for any single lithium powered mobile device rated at 4.5 volts (maximum 3 amps) or lower (Figure 22). Power Tool software and the Power Monitor hardware can be used to optimize the design and analyze the performance of mobile devices. The Power Monitor can measure data on three possible channels: Main, USB, and Auxiliary. The Main channel is what we used for our measurements. We set up a dedicated Windows platform workstation in order to achieve the optimal performance and results with the Power Monitor. A necessary part of
instrumentation set up was bypassing the battery of the mobile device and powering the device with the Monsoon Power Monitor. The setup still allows the device to continue communicating with the battery. In order to create the connection, we have to isolate the $\text{Voltage (+) terminal}$ on the battery and directly bypass the Power Monitor $V_{\text{out}}$ to the device. The circuit is completed by connecting directly to the $\text{Ground (-) terminal}$ on the battery. Once the setup is done, we are ready to start measurements.

### 6.3 Battery impact comparison

We analyzed Samsung Galaxy SIII power consumption in three configurations: idle, MPower running, daemon running. In each case, we deactivated each hardware component that could impact on battery (e.g. WiFi, GPS, Bluetooth, etc). In this way, apart from Android background services, it was possible to see only the contribution of MPower and our daemon on power consumption. Our idea was: we collect the data produced by the Monsoon Power Monitor, then we use a statistical test that compares the mean of each measurement and, finally, we find out if our daemon impact on battery is greater or smaller than MPower impact. The Monsoon Power Monitor, given a constant $\text{Voltage}$, shows either the $\text{milliAmpere}$ or the $\text{Watt}$ required by the device at every time instant. We set the Voltage to 3.7V and chose to collect the data as milliAmpere.

We followed three different approaches in order to the gathered data.

**One test of 1000 seconds**

MPower (or the daemon) are supposed to log the device state every 10 seconds. Therefore in was necessary to choose a sufficient number of logs in order to provide significant results.
We figured out one hundred as a meaningful number of logs. For this reason, we chose to run our one test of 1000 seconds for each configuration. The Monsoon Power samples, by default, the data at 5KHz; hence, we collected nearly one million and a half samples for each configuration. However, it was not possible to compare such results since their distribution was not Gaussian. Indeed, we ran the Jarque-Bera test\cite{89}, which returns a test decision for the null hypothesis that the data come from a Gaussian distribution with an unknown mean and variance, and the null hypothesis was rejected at the 5% significance level. Moreover, $p$-value of this test was very low (less than 0.0001), which indicates the statistical significance of the test. It was clear that this approach was neither feasible nor significant. Therefore, we changed approach.

30 tests of 5 minutes

We ran 30 tests of 5 minutes each for all the three configurations. Then, we summed all the samples of each tests, in order to have 30 cumulative samples of each configuration. In this way, it is more likely that such samples may represent better their distribution. After collecting the data of idle and MPower configuration, we ran again the Jarque-Bera test on the 30 cumulative samples, and the null hypothesis was accepted for both configuration. This means that there was no statistical evidence to state that the two distributions were not Gaussian. However, another problem occurred: although the samples were considered normally distributed, the \textit{two-sample t-test}\cite{90} accepted the null hypothesis (the data come from independent random samples from normal distributions with equal means and unknown variance). It was clear that MPower impact on the battery (logging device state
every 10 seconds) was so not significant to provide a statistical evidence of a different power consumption. For this reason, we pushed MPower logging service frequency, first, to 1Hz, then, to 1KHz. It was in this occasion that we found out that MPower logging service frequency cannot go further than 1Hz (Section 4.3). Therefore, we ran new tests with only MPower and set its logging service frequency to 1Hz. Indeed, it was useless to run tests with our daemon if we were not even able to prove that MPower consumes more power than idle state. Once data gathering was done, we used again the two-sample t-test and it accepted again the null hypothesis.

Neither this approach seemed to be correct. If we analyze the data collected in the measurements, we see behaviors like the one in Figure 23a and Figure 23b. These are just random samples extracted from MPower and idle state measurement, they have no statistical significance on their own. However, they are interesting since they show that, most of the times, the mobile device Current consumption is less than 5mAmpere, while we have sporadically some peaks. For sure, some of the peaks we see in Figure 23a represent MPower logging service activity.

30 tests of 5 minutes and filtering

Hence, our new approach is to filter data over a certain threshold, in order to build samples that contain only the peaks, then, as previously, sum the samples of each filtered test and, finally, build the 30 cumulative samples for each configuration. We chose 5mA as threshold, which corresponds to the 92 percentile of idle state measurements. If we now apply the two-sample t-test to the filtered cumulative samples from both MPower
(a) Sample of MPower consumption

(b) Sample of idle consumption

Figure 23: Samples of Monsoon Power Monitor measurements
and idle state measurements, the test would be biased since the threshold comes from idle
state measurements. Just for the records, we tried the two-sample t-test anyway with,
as alternative hypothesis, MPower (logging service at 1Hz) mean greater than idle state
mean, and the test rejected the null hypothesis, at the 5% significance level, with the
$p$-value equal to 0.0266. For both the cumulative samples, the null hypothesis of Jarque-
Bera test was accepted.

We sampled the daemon (at the same logging frequency of MPower) power impact with 30
tests of 5 minutes each, as usual. First of all, we test such data with Jarque-Bera test and,
again, the null hypothesis was accepted. Then, we filtered the data and assembled the 30
cumulative samples. Finally, we tested such samples with the ones filtered from MPower
measurements. The two-sample t-test rejected the null hypothesis, at the 5% significance
level, that the consumption means were equals and, therefore, accepted the alternative
hypothesis (MPower mean consumption greater than daemon mean consumption), with
the $p$-value equal to 0.0053. Finally, we ran the same test and compared the daemon
measurements with the ones from the idle state, even though we knew this result would
be biased. The two-sample t-test accepted the null hypothesis, at the 5% significance
level ($p$-value equal to 0.2863). Hence, there was no statistical evidence to state that our
daemon affects battery life.

Hence, we have shown that the mean power consumption of our daemon is smaller than
MPower mean power consumption (in Table XI we show some of the most significant tests we
have performed).
### TABLE XI: SOME SIGNIFICANT TESTS

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Filter</th>
<th>Run x time(s)</th>
<th>Test</th>
<th>$H_0$</th>
<th>$H_1$</th>
<th>Rejected</th>
<th>$p$-value</th>
<th>Sign. level</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPower (10sec)</td>
<td>None</td>
<td>No</td>
<td>1x1000</td>
<td>JB-Test</td>
<td>X is normal distributed</td>
<td>X not normal distributed</td>
<td>Yes</td>
<td>&lt; 0.001</td>
<td>5%</td>
</tr>
<tr>
<td>Idle state</td>
<td>None</td>
<td>No</td>
<td>1x1000</td>
<td>JB-Test</td>
<td>X is normal distributed</td>
<td>X not normal distributed</td>
<td>Yes</td>
<td>&lt; 0.001</td>
<td>5%</td>
</tr>
<tr>
<td>MPower (1sec)</td>
<td>None</td>
<td>Yes</td>
<td>30x300</td>
<td>JB-Test</td>
<td>X is normal distributed</td>
<td>X not normal distributed</td>
<td>No</td>
<td>0.0954</td>
<td>5%</td>
</tr>
<tr>
<td>Daemon</td>
<td>None</td>
<td>Yes</td>
<td>30x300</td>
<td>T-Test</td>
<td>X mean = Y mean</td>
<td>mean X &gt; Y mean</td>
<td>Yes</td>
<td>0.0220</td>
<td>5%</td>
</tr>
<tr>
<td>MPower (1sec)</td>
<td>Idle state</td>
<td>Yes</td>
<td>30x300</td>
<td>T-Test</td>
<td>X mean = Y mean</td>
<td>mean X &gt; Y mean</td>
<td>Yes</td>
<td>0.0053</td>
<td>5%</td>
</tr>
<tr>
<td>Daemon</td>
<td>Idle state</td>
<td>Yes</td>
<td>30x300</td>
<td>T-Test</td>
<td>X mean = Y mean</td>
<td>mean X $\neq$ Y mean</td>
<td>No</td>
<td>0.2063</td>
<td>5%</td>
</tr>
</tbody>
</table>

![Figure 24: Daemon - MPower mean timestamp interval comparison](image)

Data are represented using a logarithmic scale

#### 6.4 Daemon logging frequency

In order to find out our daemon logging frequency, we modified its code. In particular, we prevent the system to put the daemon to sleep for 10 seconds after collecting the device state. When the daemon appends data to the log file, it inserts a timestamp of each log. However, actual timestamp granularity is too coarse (i.e. seconds), therefore we modified the timestamp computation, in favor of a finer granularity (microseconds). Then, we ran our daemon for some
time (in the same conditions of the previous tests), we extracted the same number of timestamps we had extracted from MPower database (1000), and we compared them. On one hand, as stated in Section 4.1, MPower, when pushed over the edge, logs the device state, on average, every 1.25 seconds (0.80Hz), while the standard deviation is 427 milliseconds. On the other hand, it turned out that our daemon is able to collect data about the system, on average, every 671 microseconds (1.5KHz), whereas the standard deviation is 67.44 microseconds. Figure 24 shows such comparison. Here, it is clear how the Java API stack complicates and slows down MPower logging service. Therefore, if we need a high frequency log of the device or a certain component of it, we have to move the logging service outside the Android runtime environment.
CHAPTER 7

CONCLUSIONS AND FUTURE WORKS

Here we present the final considerations on the work done with this first implementation of our solution, an Android OS-level daemon able to collect the state of a device. Our daemon has effectively succeeded in logging device state, and, most important, in comparison with MPower, in having both a lower impact on device battery life and a higher log frequency.

In Section 7.1, we discuss the contribution of this work together with its limits. Then, in Section 7.2 we present the current works, while, in Section 7.3, we discuss the future works which could derive from this one.

7.1 Contributions and limits

The main contribution of this work is that we have created an enabling technology, decoupled from any other high level App but, at the same time, easily accessible, able to log the state of a device components (e.g. WiFi, GPS, Bluetooth, etc.) with a low impact on battery life and a sufficient high frequency. Then, such information may be used in many ways, like building a power model able to predict the TTL of a device battery. Moreover, nowadays there are many applications that rely on data generated by the device sensors and need to collect them as fast as possible, in order to perform a reasonably accurate data analysis. For instance, there are many Apps that, using internal or external tools, gather information about a person’s health. In these cases, it is heartily recommended to collect data provided by sensors quickly and efficiently.
A major limit of this work is that, at the moment, we are not collecting all the data that MPower collects. Indeed, we were not able to find some data, for instance about device audio. However, this was not a big deal in our case since MPower, although it collects the data shown in Table VII, does not use, in the actual version, all of them to build the power model. For instance, the CPU frequency is not used since it useless if gathered at the actual logging frequency of MPower. We have to say that it is possible to gather these missing information by using the HAL interfaces to the drivers. However, this update could threaten our daemon portability since, on different Android versions, this would imply changes in the daemon code, instead of just modifying the paths in the configuration file.

Another limit is that, in order to use our daemon, we have to use an Android Custom ROM, since we have not developed an ”ordinary” Android App. Indeed, if we want to integrate our daemon with another App, we need to install our custom version of CyanogenMod on the device. Therefore, our daemon may be considered as a useful tool to be used just for particular tasks that required system state logging.

7.2 Current works

The most immediate step is to modify the deamon so that it can collect all the data that, at the moment, it is missing. In order to achieve such goal, we need to deeply analyze Android HAL. This will give us a wider view of how Android System Services get the information they expose to higher levels. Hardware components can be accessed using the HAL libraries called \texttt{libhardware.so} and \texttt{libhardware_legacy.so}. We have already analyzed and used some interfaces of \texttt{libhardware_legacy.so}. Indeed, such library provides the header file \texttt{power.h},
which allows to get and release a wake lock. Moreover, the corresponding file `power.c` indicates which files of `Sysfs` show the wake locks requested. On the other hand, `libhardware.so` allows to get an hardware module and use its callbacks. In this way, we would be able to log the remaining information. Of course, this new implementation may have a different impact on battery since we are not reading from files anymore but we are using HAL APIs. It would be interesting to measure power consumption also in this case and compare the results with the ones we have already collected.

Another possible scenario could be a more deep integration with MPower App. At the moment, thanks to JNI, MPower is allowed to check whether our daemon is running or not. In this way, MPower can avoid to run its on logging service. We can think of other possible integration. Indeed, MPower, once it has noticed that our daemon is running, could group the log files just produced and, when certain requirements are satisfied (i.e. the device is plugged in and connected to a network via WiFi), zip and send them to its server, in order to generate a new power model. In this way we could test whether the power model generated by such data is more or less accurate than the actual one. Besides, when MPower suggests the user the possible device configurations that will permit the device battery life to live enough according to the user’s needs, it cannot change the state of all the hardware components. For this reason, it would be useful if our daemon could get a notification from MPower and apply the changes required, for instance using `libhardware.so` and `libhardwarelegacy.so`. This could help MPower accomplish better its goal of extend battery life.
7.3 Future works

Nowadays, the number of sensors that are meant to be coupled with mobile devices is rapidly increasing. Such sensors generate streams of data that have to be continuously analyzed, as efficiently as possible, i.e., in terms of performance and power consumption. An example is the field related to analysis and processing of biomedical data. In such case, it is crucial to collect data at a certain frequency in order to produce a reasonable model. Furthermore, even if the type and the purpose of sensors a user may need is strictly dependent on the personal usage, mobile devices are currently built on a predefined set of hardware components, without any possibility to be customized. Google Project Ara\cite{91} is moving in this direction, proposing a modular architecture to overcome the current mobile devices limitations. In this vision, a technology like Field Programmable Gate Array (FPGA) may help developers to process sensor data in a faster and more power efficient way with respect to a computation performed on the device CPU. Our daemon could perfectly fit with such context, since it can collect the sensors data at a sufficient high frequency in order to efficiently and punctually provide them to the FPGA for processing and analysis. This is probably the most interesting and challenging of the future works, since it is something completely innovative that aims at creating a brand new market.
CITED LITERATURE


65. NECSTLab. http://necst.it/about.


VITA

Emanuele Del Sozzo

Education

<table>
<thead>
<tr>
<th>Degree</th>
<th>Program</th>
<th>Institution</th>
<th>Location</th>
<th>Dates</th>
</tr>
</thead>
<tbody>
<tr>
<td>B.S.</td>
<td>Engineering of Computing Systems</td>
<td>Politecnico di Milano</td>
<td>Milano</td>
<td>July 2012</td>
</tr>
<tr>
<td>M.S.</td>
<td>Engineering of Computing Systems</td>
<td>Politecnico di Milano</td>
<td>Milano</td>
<td>2012-2014</td>
</tr>
<tr>
<td></td>
<td>Alta Scuola Politecnica, IX Cycle</td>
<td>Politecnico di Milano, Politecnico di Torino</td>
<td>Milano, Torino</td>
<td>2012-2014</td>
</tr>
<tr>
<td>M.S.</td>
<td>Computer Science</td>
<td>University of Illinois at Chicago</td>
<td>Chicago, IL</td>
<td>2012-2014</td>
</tr>
</tbody>
</table>